Texts in Computer Science

Marco T. Morazán

Programming-Based Formal Languages and Automata Theory

Design, Implement, Validate, and Prove

Springer

Texts in Computer Science

Titles in this series now included in the Thomson Reuters Book Citation Index!

'Texts in Computer Science' (TCS) delivers high-quality instructional content for undergraduates and graduates in all areas of computing and information science, with a strong emphasis on core foundational and theoretical material but inclusive of some prominent applications-related content. TCS books should be reasonably self-contained and aim to provide students with modern and clear accounts of topics ranging across the computing curriculum. As a result, the books are ideal for semester courses or for individual self-study in cases where people need to expand their knowledge. All texts are authored by established experts in their fields, reviewed internally and by the series editors, and provide numerous examples, problems, and other pedagogical tools; many contain fully worked solutions.

The TCS series is comprised of high-quality, self-contained books that have broad and comprehensive coverage and are generally in hardback format and sometimes contain color. For undergraduate textbooks that are likely to be more brief and modular in their approach, require only black and white, and are under 275 pages, Springer offers the flexibly designed Undergraduate Topics in Computer Science series, to which we refer potential authors.

Marco T. Morazán

Programming-Based Formal Languages and Automata Theory

Design, Implement, Validate, and Prove

Springer

Marco T. Morazán
Department of Computer Science
Seton Hall University
South Orange, NJ, USA

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-031-43972-8 ISBN 978-3-031-43973-5 (eBook)
https://doi.org/10.1007/978-3-031-43973-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Charito, Josh, Josie, Oliwia, Tiksi, Sach, and Shamil. You are among the heroes in the `FSM` *story.*

Marco

Preface

1 To Students

You are about to embark on a fascinating journey through theoretical computer science. The word *theoretical* may make the material sound intimidating, hard, and boring. Historically, there have been good reasons for a computer science student to feel that way. The primary reason is that formal languages and automata theory has been taught as a pencil-and-paper course. Students have been asked to design machines and grammars – which are essentially programs – on paper and then write theorems about their properties. Indeed, this goes against the grain of your training and expertise. After all, when you design a program, you implement it and get immediate feedback from an interpreter or compiler and from testing. This is impossible if a pencil-and-paper design is never implemented.

Why is implementation being discussed in a theoretical computer science textbook? Quite simply, formal languages and automata theory is about programming. Machines and grammars are representations of programs using an `API`. As such, they can and ought to be implemented. Once testing provides you with confidence that the program works, then you can reason formally about it and write theorems. This textbook, therefore, is based on programming and explores practical applications of theoretical concepts. In the process, you shall become a better programmer and learn the limitations of what can and cannot be done with a given `API`. This naturally culminates in discovering problems that can be solved efficiently, problems for which no efficient solution is known, and problems that cannot be solved. In no small sense, this journey takes you to the very limits of what is possible.

Programs are written in a domain-specific language embedded in `Racket` called `FSM` (**F**unctional **S**tate **M**achines). `FSM` allows you to implement the machines and the grammars you design. You may test your designs and visualize the execution of machines. In this manner, you may debug your designs

before submitting for grading and before writing a proof. Furthermore, you may implement the algorithms you design as part of your constructive proofs. If you do not know what a constructive proof is, do not worry. We shall learn about them.

In addition to making you a better programmer, this textbook ought to expand your mind. It ought to help you distinguish when a problem can and cannot be solved. Consider, for example, the problem of determining if for a given program, `P`, and its input, `w`, `P` halts on `w`. This would be an incredibly useful program to write because it would tell us if `P` ends or goes into an infinite recursion or loop. Can such a problem be solved and a program written for it? We shall discover the answer to this and many other problems through the pages of this textbook. Let us get started!

2 To Instructors

This textbook is intended as an introduction to formal languages and automata theory for upper-level undergraduates or beginning graduate students. It contains the traditional mathematical development employed when you and I took our first computational theory course. It is also quite different from such a course. Machines, grammars, and algorithms developed as part of a constructive proof are intended to be rendered as a program. Encourage your students to implement and test their designs before writing a proof. As we know, sometimes a student's design may be difficult to understand and/or have mistakes that lead them to be marked down. By rendering their machines, grammars, and constructive algorithms as proofs, this problem ought to be mitigated and provide a framework to discuss the design of a solution with a student.

In addition to formal reasoning about languages, students are also engaged in formal reasoning about the machines and grammars they design. To this end, for example, they are encouraged to implement invariant predicates for the states of machines. The `FSM` visualization tool allows students and instructors to see if the state invariants hold during machine execution. If machines are tested and state invariants are validated, then it is easier to prove that machine, `M`, for a language `L` accepts a word `w` if and only if $\texttt{w} \in \texttt{L}$ and rejects `w` if and only if $\texttt{w} \notin \texttt{L}$. Indeed, this is something that neither you nor I did in our theory courses. Keep in mind that just like in programming, those that take the time to write invariants tend to write bug-free machines. It is time well-invested given that it will reduce frustration with grading (for both you and your students) and it will make your students better programmers.

Although formal proofs are an integral part of this textbook, they are only presented when they are elucidating. In other words, we do not want to burden students with *proving the obvious* nor with low-level details that drive them away from the big picture. In this spirit, many low-level inductive proofs

are left as exercises and may be assigned if you feel that your students need more practice developing formal arguments. This is not to say that hand-waving is acceptable in lieu of formal reasoning. Students ought to develop formal reasoning skills.

Although you and I spent time drawing machines in our automata theory courses, there is no need for students using this textbook to do so. Let FSM worry about rendering the graphical representation of machines. Also, use FSM to walk through the execution of a machine. This may be done using a control view or a (traditional) graph-based view. Either view may be used to validate state invariants during execution. Feel free to use the view you are most comfortable with despite the textbook tending to use the control view.

Although there may be a learning curve for using FSM, have fun with the material and make it fun for your students. Encourage implementation and experimentation. Experience has taught me that computer science students are less frustrated and become more engaged when allowed to program. I hope you have a similar experience!

3 FSM

3.1 Installing

3.1.1 DrRacket

To program in FSM, you will need Racket and its IDE known as DrRacket. You may download it as a package from:

```
https://download.racket-lang.org/
```

After installing DrRacket, go to the Language menu and click on Choose Language..., or simply use Ctrl-L. A pop-up window will appear. Choose The Racket Language and click on OK. This tells DrRacket that you will be programming using Racket syntax.

Click on Run, and you will see the basic DrRacket interface. There are two windows. The top window is called the definitions window, and this is where programs are written. The bottom window is the interactions window, and this is where values are printed and where the programmer may interface with her program.

3.1.2 FSM

FSM is a domain-specific language embedded in Racket that requires a software package called GraphViz. GraphViz is an open-source graph

visualization software. The FSM visualization tool uses it to render a graph-based view of state machines. A description of GraphViz may be found at:

https://graphviz.org/about/

There is no need for you to learn how to program using GraphViz. The necessary programming is implemented as part of FSM. Instructions for downloading and installing GraphViz are found at:

https://graphviz.org/download/

After installing DrRacket and GraphViz, you must install FSM as a package in Racket. The needed files may be found at:

https://github.com/morazanm/fsm

To install the package using DrRacket, go to the File menu and click on Package Manager.... For Package Source, type the above url followed by .git as follows:

https://github.com/morazanm/fsm.git

Click on Install. When the package finishes installing, you may close the package manager window. You now have access to all the programming constructs that are defined by FSM.

3.2 Writing FSM Programs

FSM provides and allows a programmer to build and test regular expressions, state machines, and grammars. In addition, it may be used to render a graphical representation of a state machine and to visualize the execution of state machines. It inherits its syntax and many programming constructs from Racket. This means that you may write FSM programs using anything available in Racket. An overview of the required syntax is discussed in Chap. 1.

The following is a quick example of an FSM program:

```
#lang fsm

(define AB (concat-regexp (singleton-regexp "a")
                          (singleton-regexp "b")))

(check-equal? (printable-regexp AB) "ab")
```

This program builds a regular expression for "a" and another regular expression for "b" and defines a variable AB to store the regular expression for "a" concatenated with "b". The last line in a program is a unit test. It tests that converting AB into printable form produces the string "ab".

3.3 Bug Reporting

We are constantly trying to improve FSM. Although we hope there are none, it is practically inevitable that you may discover a bug in FSM. Kindly report bugs at:

https://morazanm.github.io/fsm/

At the top right-hand, there is a link for bug reports.

When reporting a bug, fill in as much information as possible including a description of the bug, the steps to reproduce it, and the operating system used. You may include screenshots and attach an FSM file with the code that generates the bug. Keep in mind that this tool is to report FSM bugs and not to seek help debugging your programs nor to seek help on class assignments. Discuss class assignments with your instructor.

4 The Parts of the Book

The book is divided into four parts that build on each other. Part I reviews fundamental concepts. It introduces programming in FSM and reviews program design. In addition, it reviews essential mathematical background on sets, relations, and reasoning about infinite sets. This part of the book ends with an overview of different proof techniques including those using formal logic, mathematical induction, the pigeonhole principle, contradiction, and diagonalization. Recall that this textbook aims to make your programming experience relevant as you advance. This part of the book might surprise you by using proof techniques in conjunction with programming.

Part II starts the study of formal languages and automata theory in earnest with regular languages. It first introduces regular expressions and shows how they are used to write programs that generate words in a regular language. Applications of regular expressions are discussed, and programming a password generator is explored in detail. A fascinating discovery made is the role of nondeterminism in computing. That is, nondeterminism is a language feature in FSM much like integers are a language feature in your favorite programming language. Given that regular expressions generate words, it is only natural to ask how can a machine recognize words in a regular language. This leads to the study of deterministic and nondeterministic finite-state machines. We see that nondeterminism may also be used to recognize the members of a language. In addition, a new notation to generate regular languages, called regular grammars, and a theorem used to prove that a language is not regular are presented.

Part III starts the exploration of languages that are not regular with context-free languages. It starts with context-free grammars and pushdown automata to generate and recognize context-free languages. Their equivalence

is discussed as well as properties of context-free languages. This discussion includes a theorem used to prove that a language is not context-free. This part ends with a discussion of deterministic pushdown automata and illustrates why these automatons are fundamentally different from nondeterministic pushdown automata.

Part V marks the culmination of our studies by exploring languages that are not context-free, known as context-sensitive languages. It starts by discussing the most powerful automaton known to mankind: the Turing machine. It explores the versatility of Turing machines by illustrating how to compose them. That is, it illustrates using auxiliary Turing machines much like you use auxiliary functions or methods. The discussion explores attempts to strengthen Turing machines (i.e., create more powerful Turing machines). Although all such attempts have failed to date, useful abstractions have emerged from these failed attempts that make the design, implementation, validation, and verification of Turing machines easier. It is likely refreshing to see that computer scientists can learn from failures – a discipline that is not known to publish negative results nor lessons from failed approaches. Our study then moves to grammars for context-sensitive languages, and their equivalence with Turing machines is explored. This part ends with two chapters that explore the (current?) limitations of mankind. We first study the Church-Turing thesis and define the word *algorithm*. This leads to the exploration of problems that we can postulate but cannot solve. The book ends with a brief chapter introducing complexity theory and explores the question of determining if a solution to a problem is practical.

5 Acknowledgments

This book is the product of work done by many bright computer scientists throughout my professional development dating back to my years as an undergraduate computer science student. Some of these bright computer scientists inspired me to look for a better way to teach formal languages and automata theory, while others enthusiastically embraced my view and contributed to making `FSM` a reality. As an undergraduate and graduate student, I disliked the delivery of all my automata theory and formal languages courses. I found the material fascinating – even exciting – but as a computer science student, I did not understand why everything was theoretical and why it made sense to get lower marks for trivial mistakes that would have easily been revealed using a programming language to implement and test my designs. I thank all my formal languages and automata theory professors for inspiring me to promise myself I would find a better delivery if I ever became a computer science professor. You, the reader, will now judge how well my efforts have achieved the fulfillment of that promise. I think you are in for a treat – a

programming-based formal languages and automata theory course is something I dreamed about as a student.

As a researcher specializing in programming languages, I realized that any programming-based approach to formal languages and automata theory required a domain-specific language – a tool my instructors never had. Without a doubt, my work builds on the work of `PLT` headed by Matthias Felleisen. Their work developing a programming-languages programming language, `Racket`, has made the realization of my vision practical and easier. For this, I thank my friend Matthias and my friends in `PLT` – all of whom have always happily discussed with me how to teach computer science.

I must thank my Ph.D. advisor, Douglas Troeger, for helping me realize as a graduate student that state-based machines may be verified much like we verify programs. The challenge, of course, is that students had to verify without being able to validate. From this rose my idea for a visualization tool in which we can observe if the design role of states holds during a computation.

My most heartfelt thank you is for my research assistants that bought into my vision and selflessly gave (and continue to give) themselves to the development of `FSM`. Among my former students, I must express my deepest gratitude to Joshua M. Schappel (Josh), Josephine A. Des Rosiers (Josie), and Rosario Antunez (Charito). Josh has guided, with inspiration and enthusiasm, the implementation of the `FSM` visualization tool. Josie brought to fruition `FSM`'s sensible error-messaging system. Charito was there at `FSM`'s very own big bang working long hours to get the language off the ground. I also thank my other former research students for actively contributing, using, and suggesting improvements to `FSM` including Shamil Dzhatdoyev and Sachin Mahashabde (Sach). Finally, I must also extend a heartfelt thank you to my current research students. Oliwia Kempinski is extending `FSM` with computation graphs, and Tijana Minic (Tiksi) is adding machine-transformation visualization tools. Their work will soon be integrated into `FSM`. As a whole, this group of students joined me because they wanted to change the world and make a positive contribution. I believe we have done so! ♡

South Orange, NJ, USA Marco T. Morazán

Contents

Part I
Fundamental Concepts

Chapter 1
Introduction to FSM

Historically, formal languages and automata theory courses have been theoretical pencil-and-paper courses. Students design algorithms in theory (i.e., without implementing them) and write theorems based on the algorithms they design. If there is a bug in the algorithm, then it is commonly the case (especially among students) that the bug is not discovered and there is, of course, also a bug in the proof of a theorem. This truly goes against the grain of a computer science education. As a computer science student, you have been trained to *design* and *implement* algorithms. Unit testing and runtime bugs give you immediate feedback on your implementation providing the opportunity to make corrections before submitting work for grading. This is rather difficult to do if algorithms are only designed and never implemented.

This textbook takes a novel approach. It complements pencil-and-paper artifacts (e.g., like writing a proof) with programming. It uses a domain-specific language called `FSM` (**F**unctional **S**tate **M**achines). `FSM` provides readers of this textbook with the ability to design, program, test, and debug algorithms before writing theorems or submitting for grading. This brings to bear your training and experience as a student of computer science.

To start, let us learn the basics of `FSM` syntax. If you have previously programmed in `Racket` (or any programming language in the `Lisp` family), then you are already mostly familiar with the basics of `FSM` syntax.

6 FSM Syntax

6.1 Boilerplate Code

To write `FSM` programs, the first line in every program must be:

```
#lang fsm
```


M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_1

This line informs DrRacket that the programming language used is FSM. In addition to FSM primitives, this language provides the testing facilities from rackunit. You may search DrRacket's Help Desk for rackunit to learn about the facilities provided to write unit tests. Throughout this book, a program is never complete without unit tests.

6.2 Value Expressions

All programs are written using expressions. Racket includes primitive values such as numbers, symbols, Booleans, and strings. All of these are also expressions. They evaluate to the constant that they represent. For example, type this program in DrRacket's definitions window:

```
#lang fsm

"I love FSM!"
42
'mom
#t
```

This program consist of four expressions. The first is a string. All strings go inside quotes. The second is a number. The third is a symbol. One or more alphanumeric symbols with no spaces preceded by ' is a symbol. The fourth is a Boolean. In this case, it is the value true. The value false is denoted by #f. Boolean values may also be written as #true and #false. Running the program produces the following result in the interactions window:

```
"I love FSM!"
42
'mom
#t
```

Each expression is evaluated and its value returned. In this example, the returned values are printed in DrRacket's interactions window.

6.3 Primitive Functions and Application Expressions

There is a plethora of functions provided by FSM – just the set of Racket functions is too big to exhaustively cover here. Primitive functions include arithmetic, Boolean, string, and symbol operators. When unsure if a function exists in FSM, you can search for it by going to the Help menu and then the Help Desk.

Function applications are always written inside parenthesis in prefix notation:

```
( <function> <expression>* )
```

To apply a function, put, inside parenthesis, an expression for the function (e.g., the function's name) followed by the arguments.[1] Type `(+ 1 2 3)` at the prompt in the interactions window, and hit return. The expression is evaluated, and its value is printed before returning the prompt:

```
> (+ 1 2 3)
6
```

The following is the result of evaluating an expression to test if `'a` is equal to `'b`:

```
> (eq? 'a 'b)
#f
```

Remember that inside parenthesis, the function always goes first followed by the arguments, if any, separated by at least one blank space.

6.4 Lists

Lists are primitive values in `FSM`. The constructor for a list is `cons`, and it takes as input two arguments: the first element of the list and the rest of the list. The second argument to `cons` must be a list. The empty list is denoted by `'()` or `null`. For instance, the following builds a list with two strings:

```
> (cons "Hello" (cons "world!" '()))
'("Hello" "world!")
```

Function composition (as the nested `cons` above) are common practice in `FSM`.

The selector functions for lists are `first` and `rest` (a.k.a. `car` and `cdr`). The following is an interaction extracting the first element of a list:

```
> (first (cons 1 (cons 10 '())))
1
```

The value 1 is returned because 1 is the first element of the list given to `first`. This is an interaction for extracting the rest of a list:

```
> (rest (cons #t (cons "hi" '())))
'("hi")
```

The list containing only `"hi"` is returned because that is the rest of the list given to `rest`. Here, the quote, ', is not indicating that the value returned is a symbol. When a quote appears before an opening parenthesis, it indicates that the value is a list. In this manner, a function application, like `(f x)`, is easily distinguished from the list, `'(f x)`, containing `f` and `x`.

[1] Observe that one or more blank spaces (not a comma) separate arguments.

There are several useful list-manipulating functions like:

```
          (empty? L): evaluates to #t if L is empty and
                      #f otherwise (same as null?)
          (length L): evaluates to the number of elements in L
       (append L1 L2): evaluates to a list that starts with the
                      elements of L1 and ends with the
                      elements of L2
(remove-duplicates L): evaluates to a list with any repetitions
                      in L removed
            (last L): evaluates to the last element of L
```

There are many more useful list-manipulating functions. You are referred to the `Help Desk` to search for functions you need and may already be part of `FSM`.

6.5 List Abbreviations

Building lists using `cons` can be tedious. For every list element, `cons` must be typed. Imagine doing that for a list with 20 elements. To avoid all this repetitive typing, there are three shorthand list constructors: `'`, `list`, and `` ` ``.

A *quoted list* has the elements listed inside parenthesis preceded by a `'`. For example, the following expression evaluates to a list with the digits 1, 2, and 3:

```
'(1 2 3)) = (cons 1 (cons 2 (cons 3 '())))
```

Using a quoted list eliminates the need to repeatedly write `cons` to construct a list. It is important to note that nothing after the quote (inside the parenthesis) is evaluated. That means all the listed elements are literal values in the constructed list. This is important because there can be no expressions that need to be evaluated inside the parenthesis.

To create a list with the values of evaluated expressions, we may use `list`. The number of arguments is arbitrary. If no arguments are provided, `list` returns `'()`. The following are sample expressions using `list`:

```
(list) = '()

(list (+ 3 4) (list) #f) = (cons 7 (cons '() (cons #f '())))
```

Use `list` when you want all the expressions to be evaluated.

The third shorthand is a *quasiquoted list*. A quasiquoted list is a combination of `'` and `list`. Instead of preceding the opening parentheses with a `'`, it is preceded with `` ` ``. The `` ` `` indicates that some subexpression inside the parenthesis may have to be evaluated. To indicate that an expression needs to be evaluated, it must be preceded by a `,`. If there are no expressions preceded

by a comma inside the parenthesis, then ` and ' yield the same value when evaluated. To illustrate the use of quasiquote, consider the following code:

```
`((+ 3 4) ,(+ 3 4)) = (list '(+ 3 4) 7)
```

It is important to keep in mind that only the expressions preceded by a comma are evaluated.

6.6 Conditional Expressions

In order for a program to make a decision, conditional expressions are needed. In FSM, there are two types of conditional expressions available: if-expressions and cond-expressions. An if-expression is best used when there is a binary decision to be made (i.e., there are only two conditions). The syntax for an if-expression is:

```
(if <expression> <expression> <expression>)
```

The first expression is the condition that is tested. That is, it is an expression that evaluates to a Boolean. If the condition evaluates to true, then the second expression, called the then-expression, is evaluated to obtain the value of the if-expression. Otherwise, the evaluation of the third expression, called the else-expression, is evaluated to determine the value of the if-expression. For instance, consider the following expressions:

```
(if (odd? x) 1 0)

(if (or (< x -5) (> x 5))
    (f x)
    (error (format "Invalid value for x: ~s" x)))
```

The first expression evaluates to 1 if x is odd and evaluates to 0 if x is even. The second expression evaluates to the value of (f x) if $|x| > 5$ and throws an error otherwise. An error is thrown using error that takes an input a string for the error message. In the example above, the given input to error is formatted by substituting the i^{th} ~s with the value of the i^{th} expression after the string. In the example above, the only (i.e., the first) ~s is substituted with the value of x (i.e., the value of the first expression after the string).

A cond-expression is best used when there are more than two conditions. For the purposes of this textbook, the syntax for a cond-expression is:

```
( cond [ <expression> <expression> ]+
       [ else <expression>] )
```

A cond-expression has zero or more cond-stanzas followed by an else-stanza. A cond-stanza consists of two expressions inside square brackets. The first

is a condition, and the second, when evaluated, is the value of the `cond`-expression when the condition holds. The `else`-stanza has a single expression for the default value of the `cond`-expression. Stanzas are processed from top to bottom. If the condition for a stanza evaluates to true, then the stanza's second expression is evaluated to obtain the `cond`-expression's value. If the condition for a stanza evaluates to false, then the next stanza is processed. If all the conditions evaluate to false, then the `cond`-expression's value is the default value. For example, consider the following expression:

```
(cond [(= x 0) 0]
      [(< x 0) (* -1 x)]
      [else x])
```

Assuming `x` is a number, when this expression is evaluated, it returns the absolute value of `x`.

6.7 Defining Local Variables

There are two types of expressions for defining local variables: `let` and `let*`. The syntax for a `let`-expression is:

```
(let [(<identifier> <expression>)*] <expression>)
```

A `let`-expression has zero or more local definitions inside square brackets. Each definition has an identifier and an expression inside parenthesis. The identifier is the name of the local variable declared, and the expression, when evaluated, provides the value of the variable. The expression after the `let`-expression's declarations is the body. The evaluation of the body is the value of the `let`-expression. The local variables declared are only valid in the body and shadow any previous declaration or definition. For example, consider the following expression:

```
(let [(x 5)]
  (let [(x 2)
        (y (+ x x))]
    (* x y)))
```

This `let`-expression declares a local variable `x` as 5. Its body is a `let`-expression. The inner `let`-expression declares two local variables: `x` as 2 and `y` as the sum of `x` and `x`. The value of `y` is 10 because the `x` in scope is the one that is 5 (the `x` that is 2 is only valid in the body). The value of the let, `(* 2 10)`, is 20 (the outer `x` is shadowed by the inner `x`).

The syntax for a `let*`-expression is:

```
(let* [(<identifier> <expression>)*] <expression>)
```

A `let*`-expression has zero or more local definitions inside square brackets and a body. The evaluation of the body is the value of the `let*`-expression.

The local variables are valid in any subsequent local variable declaration and in the body. A local declaration shadows any previous declaration or definition. For example, consider the following expression:

```
(let [(x 5)]
  (let* [(x 2)
         (y (+ x x))]
    (* x y)))
```

In this expression, the value of `y` is 4 because the declaration of `x` as 2 is in scope. Therefore, the value of the expression is 8. In contrast, consider the following expression:

```
(let [(x 5)]
  (let* [(y (+ x x))
         (x 2)]
    (* x y)))
```

The value of this expression is 20, because the `x` in scope is the one declared as 5.

6.8 Functions as Values

In `FSM`, functions are first-class values. This means that a function is a value just like numbers and strings are values. Among other things, this provides us with the power to provide functions as arguments to other functions and to return a function as the value of evaluating a function.

6.8.1 Lambda Expressions

To create a function value, use a λ-expression. The syntax for a λ-expression is:

```
  ( lambda ( <identifier>* ) <expression> )
or
  ( λ ( <identifier>* ) <expression> )
```

To type λ in `DrRacket`, use `Ctrl-\`. These equivalent expressions create a nameless function that has as zero or more parameters corresponding to the given identifiers. The expression is the body of the function. When the created function is given the appropriate arguments, the body is evaluated to return the value of the function. Consider, for example, the following λ-expression s:

```
(λ (x) (* 5 x))

(λ (l1 l2) (append (rest l1) l2))
```

The first creates a function that has a parameter `x` and that multiples `x` by 5. The second creates a function with two parameters, `l1` and `l2`, and that appends the rest of the first given list and the second given list. It is important to note that functions created using a λ-expression may not be recursive because they are nameless. There is no way to call a function recursively unless it has a name. Use a λ-expression when a new function is needed, the function is not recursive, and the function is only needed once.

6.8.2 Higher-Order Functions

Higher-order (or abstract) functions have as input at least one function. They provide programmers with powerful abstractions that make their code more elegant, easier to understand, and more concise. Two abstract functions extensively used in this textbook are `map` and `filter`.

The function `map` takes as input a function, `x` → `y`, and a, `(listof x)`, list of `x` values. It applies the given function to every element in the given list and returns a list of the results:

`(map f (list` x_0 x_0 ... x_{n-1}`))` `=` `(list (f` x_0`) (f` x_1`) ... (f` x_{n-1}`))`

The following expressions are examples of `map` in practice:

```
(map add1 '(1 2 3))

(map (λ (x) (* 3 x)) '(7 8 9))
```

The evaluation of the first expression applies `add1` to every element of the given list and returns `'(2 3 4)`. The second expression's evaluation applies a function that triples its input to every element of the given list and returns `'(21 24 27)`.

The function `filter` takes as input a predicate, `x` → `Boolean`, and a, `(listof x)`, list of `x` values. It applies the given predicate to all the elements of the given list and returns a list of the elements that satisfy the predicate. For instance, consider the following expressions:

```
(filter even? '(1 2 3 4 5 6))

(filter (λ (x) (= (remainder x 5) 0)) '(1 2 25 3 10 7))
```

The first expression evaluates to, `'(2 4 6)`, a list containing the even elements in the given list. The second expression evaluates to, `'(25 10)`, a list containing the multiples of 5 in the given list.

There are other very useful higher-order functions defined for programmers. These include, for example, `andmap`, `ormap`, `foldl`, and `foldr`. You are strongly encouraged to search for these in the `Help Desk` and become familiar with them.

6.9 For Loops

A recursive function makes recursive calls explicit in the code. Loops, as you are likely to know, make recursive calls implicit. There are two types of `for`-loops that we may use in `FSM`: `for` and `for*`. Each has a keyword identifying the type of loop, one or more comprehension clauses, and an expression for the loop's body. A comprehension clause declares a variable for the sequence of values to iterate over. For each iteration step, the variable takes on the next value in the sequence. The difference between a `for`-loop and a `for*`-loop is the scope of comprehension variables. In a `for`-loop, the scope is the body of the `for`-loop. In a `for*`-loop, the scope is all the following variable definitions and the body of the `for*`-loop.

For each value a comprehension variable takes on, the body of the loop is evaluated. The type of loop used defines how the values obtained from evaluating the loop's body are combined. There are 12 types of loops:

`for/list`, `for*/list`	creates a list of all the values
`for/and`, `for*/and`	`and`s all the values
`for/or`, `for*/or`	`or`s all the values
`for/sum`, `for*/sum`	adds all the values
`for/product`, `for*/product`	multiplies all the values
`for/string`, `for*/string`	appends all the strings

The body of the loop must always evaluate to a value of the expected type. Otherwise, an error is thrown. For example, the body of a `for/string` must always evaluate to a string, and the body of a `for/and` must always evaluate to a Boolean.

To illustrate the use of a `for`-loop, consider the following expression:

```
(for/list ([v1 '(1 2 3)] [v2 '(4 5 6)])
  (* v1 v2))
```

The loop traverses both lists at the same time with `v1` taking the values 1, 2, and 3 and `v2` taking on the values of 4, 5, and 6. The list produced is:

```
(list (* 1 4) (* 2 5) (* 3 6))
```

In contrast, consider using a `for*`-loop:

```
(for*/list ([v1 '(1 2 3)] [v2 '(4 5 6)])
  (* v1 v2))
```

For every value, `v1`, in the first comprehension, the values in the second comprehension are traversed. The list produced is:

```
(list (* 1 4) (* 1 5) (* 1 6) (* 2 4) (* 2 5)
      (* 2 6) (* 3 4) (* 3 5) (* 3 6))
```

Finally, a `for`-loop stops when the end of any sequence is reached. Whatever remains of the other sequences is ignored. For example, consider the following expression:

```
(for/list ([v1 '(1 2)] [v2 '(4 5 6 7 8 9)])
  (* v1 v2))
```

The elements of the comprehensions are traversed until the elements of the first are finished. The elements `'(6 7 8 9)` from the second comprehension are not traversed. The resulting list is:

```
(list (* 1 4) (* 2 5))
```

6.10 Writing Unit Tests

A program is never considered complete without unit tests. To write unit tests, we mostly use `rackunit`'s `check-equal?`. For the purposes of this textbook, the required syntax is:

```
(check-equal? <expression> <expression>)
```

The first expression is the value you are testing. It usually is the result of a function call. The second expression is the expected value. If all tests pass after running a program, no report is generated. If tests fail, then a report for each failed test is generated in `DrRacket`'s interactions window.

Consider, for example, running the following program:

```
#lang fsm

(check-equal? (= 6 6) #t)
(check-equal? (* (+ 2 3) (/ 20 2)) 50)
(check-equal? (string-length "FSM") 4)
```

The result in the interactions window is:

```
--------------------
 FAILURE
name:       check-equal?
location:   testing-example.rkt:7:0
actual:     3
expected:   4
--------------------
```

The first two tests are successful, and no report is generated for them. The third test fails, and a failure report is generated. It includes the test, `check-equal?`, that fails and the name of the file, `testing-example.rkt`, containing, `7:0`, the line number and the position within the line of the failed test; the value, `3`, returned by the first expression in the test; and the value, `4`, of the second expression in the test. It is straightforward to see why the test failed: the length of `"FSM"` is not 4 but 3.

6.11 Definitions

In order for a programming language to be useful, it must allow programmers to design and define their own functions. Definitions in FSM are made using the following syntax:

```
( define <identifier> <expression> )
( define ( <id> <id>* )
  ( define ( <id> <id>* ) <expr>+ )*
  <expr>+ )
```

The first form is used to define constants. The definition of a constant consists of an identifier (i.e., a variable name) and an expression for the value of the constant. The second form is used to define named functions. A named function definition first has, inside parenthesis, an identifier for the name of the function followed by zero or more identifiers for the parameters. This is called the function header. The function header is followed by zero or more local function definitions. The local function definitions, if any, are followed by one or more expressions (typically one in this textbook) for the body of the function. For the purposes of this textbook, the value of a function is obtained by substituting the values of the parameters in the body of the function and evaluating the resulting expression. This model of evaluation is only valid if you are disciplined enough to not use assignment statements. Consider the following program:

```
#lang fsm

(define X 220)

(define Y -10)

;; number number → number
;; Purpose: Compute f(a, b) = 2a + b
(define (f a b) (+ (* 2 a) b))

(check-equal? (f 0 0) 0)
(check-equal? (f X Y) 430)
```

This program defines to constants: X as 220 and Y as -10. In addition, it defines a function called f. As comments, it states the signature and the purpose of the function.[2] The function takes as input two numbers and returns a number. It has two parameters, a and b, in accordance with the signature. There are no local function definitions. The function's body doubles a and adds b to this value. The tests validate that the proper value is computed. It is worth noting that the function definition syntax above is syntactic sugar to eliminate explicitly typing a λ-expression. The definition of f above is exactly the same as this desugared definition:

[2] Anything after a ; on a single line is a comment.

```
(define f (λ (a b) (+ (* 2 a) b)))
```

This should not be too surprising. Recall that in FSM functions are first-class values. You may use the sugared or the desugared syntax to define functions.

Named functions may be recursive. For example, this is a function to count the number times a given x value occurs in a given list:

```
;; X (listof X) → number
;; Purpose: Count the number of times the given value
;;          occurs in the given list
(define (count-occurrences x L)
  (cond [(empty? L) 0]
        [(equal? x (first L))
         (add1 (count-occurrences x (rest L)))]
        [else (count-occurrences x (rest L))]))

(check-equal? (count-occurrences "hi" '()) 0)
(check-equal? (count-occurrences ' x '(a b x g x h)) 2)
(check-equal? (count-occurrences '(1 2)
                                 '((w q) (9 -3) (1 2)))
              1)
(check-equal? (count-occurrences #f '(#t #t #t)) 0)
```

By using the higher-order function foldl, the need to explicitly write the recursion may be eliminated. A function to update the value of an accumulator and the initial value of the accumulator are needed to use foldl. We need a function that adds one to the accumulator when a list element is equal to x and that adds nothing to the accumulator otherwise. The program for count-occurrences may be refactored as follows:

```
;; X (listof X) → number
;; Purpose: Count the number of times the given value
;;          occurs in the given list
(define (count-occurrences x L)
  (foldl
    ;; X natnum → natnum
    ;; Purpose: Return the new value of the accumulator
    ;;          for the given X value
    (λ (an-x acc) (if (equal? x an-x) (add1 acc) acc))
    0
    L))

(check-equal? (count-occurrences "hi" '()) 0)
(check-equal? (count-occurrences ' x '(a b x g x h)) 2)
(check-equal? (count-occurrences '(1 2)
                                 '((w q) (9 -3) (1 2)))
              1)
(check-equal? (count-occurrences #f '(#t #t #t)) 0)
```

Fig. 1 The general design recipe for functions

1. Outline the representation of values.
2. Outline the computation.
3. Write the function's signature and purpose.
4. Write the function's header.
5. Write unit tests.
6. Write the function's body.
7. Run the tests and, if necessary, redesign.

The function given as input to `foldl` takes as input a value of type `X`, for an element of `L`, and, for the accumulator value, a natural number. It returns a natural number for the number of occurrences. The body is a conditional expression that returns the accumulator's new value. If a list element, `an-x`, is equal to `x`, then the new value of the accumulator is the accumulator plus one. Otherwise, it is the accumulator. The initial value of the accumulator is 0, and the list that `foldl` traverses is `L`. When `foldl` finishes traversing the list, it returns the value of the accumulator.

7 Designing Functions

7.1 The Design Recipe

There are steps you may follow to systematically design and implement functions. These steps assist you in going from a problem statement to a well-designed and tested program. They ought to help eliminate much of the guess work students engage in to write programs. This does not mean that trial and error is eradicated from the programming process. Instead, it means that designs are tried, not random code, until a satisfactory solution to the problem is found.

Figure 1 displays the general steps for function design and implementation. The first step asks the programmer to define the data types that need to be manipulated. The second step asks to outline how the problem is solved. The third step requires developing and writing as comments the function's signature and purpose. In the fourth step, the function's header is written. The number of parameters must correspond to the number of input types in the signature. Once the function header is written, the fifth step requires writing unit tests. In general, it is wise to write unit tests before developing the function's body because writing tests may provide insights into

the expression needed for the body. For the sixth step, the function's body is developed. Finally, step 7 has programmers run the tests and redesign if any tests fail. Redesign, in this context, means checking and improving the answers developed for each step of the design recipe.

7.2 Scaling a Binary Tree

To illustrate the design recipe in action, let us develop a function to scale a binary tree of numbers. We shall do so showing the results of each step of the design recipe.

7.2.1 Representation

A binary tree of numbers may be empty, a leaf, or an interior node with a number and two subtrees. A representation must be defined because binary trees of numbers are not a defined type in FSM. Observe that according to the description, there are three subtypes. This means that a data definition with three subtypes is needed. We can define a binary tree of numbers as follows:

```
;; A binary tree of numbers, (btof number), is either:
;;  1. '()
;;  2. number
;;  3. (list number (btof number) (btof number))
```

The first subtype represents the empty binary tree of numbers. The second subtype represents a leaf in a binary tree of numbers. The third represents an interior node as a list with three elements: a number and two binary trees of numbers.

Clearly defining the data to manipulate is important because it provides insights into the shape of a function. That is, the structure of the data *suggests* the structure of a function to process the data. A data definition with subtypes suggests that a conditional expression is needed to distinguish among the subtypes. The number of conditions equals the number of subtypes. The solution for each subtype is designed one at a time and independent of the other subtypes. The solution's structure for a given subtype is suggested by the structure of the subtype. Figure 2 displays the template for functions on a binary tree of numbers. The signature outlines that at least one input must be a `(btof number)`. This fact is captured by the function header that has a parameter, `a-bt`, for a binary tree of numbers. The function template's body is a `cond`-expression with three stanzas – one for each subtype. If the given binary tree is `'()`, there is no structure to suggest the solution's structure (ergo, the ... in the first stanza). If the given binary tree is a number, then it is a leaf. This suggests that the solution requires calling a function to process

Fig. 2 The template for functions on a binary tree of numbers

```
;; Template for Functions on a Binary Tree of Numbers
;;; (btof number) ... → ...
;;; Purpose: ...
;(define (f-on-bt a-bt ...)
;  (cond [(empty? a-bt)  ...]
;        [(number? a-bt) ...(f-on-number a-bt)...]
;        [else ...(f-on-number (first a-bt))...
;              ...(f-on-bt (second a-bt))...
;              ...(f-on-bt (third a-bt))...]))
;
;;; Tests
;(check-equals? (f-on-bt empty ...) ...)
;(check-equals? (f-on-bt number ...) ...)
;(check-equals? (f-on-bt (list number ... ...) ...) ...)
     ⋮
```

a number. If the given binary tree is not empty and is not a leaf, then it is an interior node. This suggest that the solution is obtained by processing a number and the two subtrees. Observe that the two subtrees are the same type of data as the given input and, therefore, are processed that same way. In general, a self-reference in a data definition means a recursive call in the function template. A binary tree of numbers has two self-references, and, therefore, the function template has two recursive calls. Finally, at a minimum, there needs to be one test for each subtype. This is reflected by the three explicit tests in the function template. Take a moment to appreciate what has been achieved. The function template is your road map for code development. More often than not, coding means specializing the function template for the problem being solved.

7.2.2 Design Idea

Ask yourself how the given binary tree of numbers may be scaled. Reason about each subtype independently. If the given binary tree is empty, then there is nothing to scale, and the resulting tree is empty. If the given binary tree is a leaf, the function template suggests calling a function on a number. To scale a leaf, it is multiplied by the given scalar. That is, the number-processing function * is called with the given binary tree and the scalar. If the given binary tree is an interior node, the function template suggests calling a number-processing function for the root value and recursively processing the two subtrees. This means calling * with the root value and the given scalar, making a recursive call with the left subtree and the scalar, and making a recursive call with the right subtree and the scalar.

7.2.3 Signature, Purpose, and Function Header

The function needs as input a `(btof number)` and a number and returns a `(btof number)`. The purpose is to scale the given binary tree by the given scalar. The signature, purpose statement, and function header are:

```
;; (btof number) number → (btof number)
;; Purpose: Scale the given (btof number) by the
;;          given scalar
(define (scale-bt a-bt k)
```

Observe that the number of parameters equals the number of inputs defined by the signature.

7.2.4 Tests

The tests validate that the function works for each of the subtypes. The tests are:

```
;; Tests
;; empty bt tests
(check-equal? (scale-bt '() 10) '())
;; leaf bt tests
(check-equal? (scale-bt -50 2) -100)
(check-equal? (scale-bt 40  8) 320)
;; interior node bt tests
(check-equal? (scale-bt (list 10 '() (list -8 -4 '())) -2)
              (list -20 '() (list 16 8 '())))
(check-equal? (scale-bt (list 0
                              (list 1 2 3)
                              (list 4
                                    (list 5 '() '())
                                    (list 6 7 8)))
                        3)
              (list 0
                    (list 3 6 9)
                    (list 12
                          (list 15 '() '())
                          (list 18 21 24))))
```

You are strongly encouraged to thoroughly test your functions. Write tests for valid inputs that you or others may suspect break your code. In this manner, you and the readers of your code can feel cautiously optimistic that the code works. You should certainly not limit yourself to one test for each subtype that has variety.

7.2.5 Function Body

The function body is obtained by specializing the conditional in the function template:

```
(cond [(empty? a-bt)  a-bt]
      [(number? a-bt) (* k a-bt)]
      [else (list (* k (first a-bt))
                  (scale-bt (second a-bt) k)
                  (scale-bt (third a-bt) k))])
```

Observe that the solution implemented for each subtype adheres to the design idea.

7.2.6 Running the Tests

The complete program is displayed in Fig. 3. Run the program and make sure all the tests pass.

Being disciplined about following the steps of the design recipe is an effective way to organize your thoughts. In addition, it provides a framework for discussing your design with your instructor and your colleagues. It may take time to develop such discipline, but it is time well-invested. You, your instructors, and your school/professional colleagues will always be grateful for well-designed, well-documented, and thoroughly tested programs.

1 Design and implement a recursive function to insert in the right place a number in a sorted list of numbers. Make sure to follow all the steps of the design recipe.

2 Design and implement a function to find the longest string in a list of strings. Make sure to follow all the steps of the design recipe.

3 Design and implement a function that takes as input two natural numbers greater than or equal to 2, `a` and `b`, and that returns the greatest common divisor of the two given numbers. Make sure to follow all the steps of the design recipe.

4 Design and implement a function that merges to sorted lists of numbers into one sorted list of numbers. Make sure to follow all the steps of the design recipe.

5 Design and implement a function that takes as input a natural number and that returns the sum of the natural numbers in `[0..n]`. Make sure to follow all the steps of the design recipe.

Fig. 3 A program to scale a binary tree of numbers

```
#lang fsm

;; (btof number) number → (btof number)
;; Purpose: Scale the given (btof number) by the given scalar
(define (scale-bt a-bt k)
  (cond [(empty? a-bt) a-bt]
        [(number? a-bt) (* k a-bt)]
        [else (list (* k (first a-bt))
                    (scale-bt (second a-bt) k)
                    (scale-bt (third a-bt) k))]))

;; Tests
(check-equal? (scale-bt '() 10) '())
(check-equal? (scale-bt -50 2) -100)
(check-equal? (scale-bt 40  8) 320)
(check-equal? (scale-bt (list 10 '() (list -8 -4 '())) -2)
              (list -20 '() (list 16 8 '())))
(check-equal? (scale-bt (list 0
                              (list 1 2 3)
                              (list 4
                                    (list 5 '() '())
                                    (list 6 7 8)))
                        3)
              (list 0
                    (list 3 6 9)
                    (list 12
                          (list 15 '() '())
                          (list 18 21 24))))
```

6 Design and implement a function using `ormap` to determine if any number in a list of numbers is prime. Make sure to follow all the steps of the design recipe.

7 Design and implement a function using `filter` to extract the strings with a length greater than 5 from a list of strings. Make sure to follow all the steps of the design recipe.

8 Design and implement a function using `map` to add a blank to the front of every string in a list of strings. Make sure to follow all the steps of the design recipe.

9 Design and implement a function that takes as input a list of numbers and a number, `i`, and that using `foldr` extracts all the numbers in the list less than or equal to `i`. Make sure to follow all the steps of the design recipe.

10 Design and implement a nonrecursive function that takes as input a list of numbers and that returns a list of the even numbers in the given list doubled. Make sure to follow all the steps of the design recipe.

8 FSM Basics

FSM is designed for the automata theory and formal languages classroom. Such a course revolves around state machines to carry out computations (e.g., determining if a given word is in a given language) and grammars to generate the words in a given language. You shall learn more about these as you progress in this textbook. This section presents constants and data definitions that you ought to be aware of as you begin. Do not worry if you do not perfectly understand what everything is the first time you read this section. The presentation here is just to start getting you familiar with terms this textbook uses.

8.1 FSM Constants

There are two constants that are useful to know before writing FSM programs:

`BLANK` Denotes a blank space in an input tape
`EMP` Denotes the empty word (i.e., a word of length 0)

8.2 FSM Data Definitions

The following are some important FSM data definitions:

`alphabet` A list of lowercase symbols of length 1 not including `EMP`.
`word` A nonempty `(listof symbol)` from an alphabet.
`nts` A set of nonterminal symbols. Each nonterminal symbol is denoted by an uppercase English letter: `[A..Z]`.
`state machine` A state machine is either:

- A deterministic finite automaton (`dfa`)
- A nondeterministic finite automaton (`ndfa`)
- A pushdown automaton (`pda`)
- A Turing machine (`tm`)

- A Turing machine language recognizer (`tm-language-recognizer`)
- A multitape Turing machine (`mttm`)
- A multitape Turing machine language recognizer (`mttm-language-recognizer`)

`grammar` A grammar is either:

- A regular grammar (`rg`)
- A context-free grammar (`cfg`)
- A context-sensitive grammar (`csg`)

Chapter 2
Essential Background

You are embarking on a journey to explore some of the most fundamental questions in computer science and, perhaps surprisingly to you, in nature. Take a moment to look around, and you find computation everywhere. Why is computation so ubiquitous? The most basic answer to that question is because problem-solving is essential for life and to our lives. People make a living calculating the price of goods, predicting the price of stocks, and making a family budget. Doctors take as input symptoms and blood work results to formulate a treatment. Evolution decides which traits are desirable in future generations. Modern `mRNA` vaccines program the immune system to produce antibodies for a specific virus. Indeed, in human life and for human life, we find computation everywhere.

9 Some Fundamental Questions

You may naturally wonder what some of the fundamental questions of computer science and, ergo, nature are. Perhaps, the most obvious one is: what is an algorithm? This may seem like a silly question to an advanced computer science student because in all likelihood, you have implemented algorithms in most of your courses. Ask around in your classroom for a definition for an algorithm. More likely than not, you will get an array of different answers. You can think of this question in a different manner. If you know what an algorithm is, then can you explain what is not an algorithm? We shall explore models of computation that allow us to formally define what "algorithm" means.

Another fundamental question we may naturally ask is: what can be computed? Put differently, is there an algorithm to solve every problem that can be posed? There's that word algorithm again, but let us try to address the

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_2

importance of this question. If at a job you are given a problem specification and are asked to implement a solution in the form of a program, how do you know such a development is feasible? If the problem cannot be solved, then it is utter nonsense (and a monumental waste of time) to try to solve the problem. We need to be aware of how to prove that a problem is unsolvable. Proving that a problem is solvable is, in principle, fairly easy. You present a solution in the form of a program (never mind the program may be difficult to write). Proving that a problem is unsolvable requires a deeper understanding of what an algorithm is and a `reduction proof` – a proof technique that we shall study.

A third fundamental question is: when is an algorithm practical? In past courses, you may have studied polynomial time and exponential time algorithms. In general, polynomial time algorithms are those that are considered practical, that is, they run in a feasible amount of time. Exponential time algorithms are not considered practical because they may take an unfeasible amount of time to run. An algorithm that takes hundreds of years to compute an answer is clearly not practical. Are there characteristics that define these sets of algorithms?

10 Automata Theory

Automata theory is concerned with the mathematical properties of computation models. It helps us understand what can and cannot be computed with a given model. You may think of this as programming using an `API`. Given an `API`, there are problems that may be solved with it, and there are problems that cannot be solved with it.

For instance, consider the following mathematical functions:

```
g(x) = g(x+1)

f(x, y) = 42
```

The value of `f(0, 50)` is clearly 42. We also have that the value of `f(0, g(8))` is 42. Can you implement these functions as a program? In `Java`, you may attempt to write methods that looks like this:

```
int g(int x)
{ return(g(x++)); }

int f(int x, int y)
{ return(42); }

void main(String[] args)
{
  System.out.println(f(10, 15));
  System.out.println(f(10, g(8)));
}
```

When you run the program, 42 is printed for the first call to `f`. Unfortunately, the second call to `f` goes into an infinite recursion, and the program bombs. In `Haskell`, the functions may be implemented as follows:

```
g:: Int -> Int
g x = g x+1

f:: Int -> Int -> Int
f x y = 42

main :: IO ()
main =  do
print(f 10 15)
print(f 10 (g 8))
```

In `Haskell`, both calls to `f` print 42, and the program exits without an error. If you are not familiar with `Haskell`, you may be asking yourself what is going on. The difference is the model of computation `Java` and `Haskell` use. `Java` implements the most common model of evaluation among modern programming languages. It is called eager evaluation – all arguments are evaluated before a function call is made. Therefore, evaluating `f(10, g(8))` causes the `Java` program to go to into an infinite recursion and abnormally terminate. In contrast, `Haskell` implements a model called lazy evaluation – function calls are made before arguments are evaluated, and an argument is evaluated only if necessary. This is why evaluating `f 10 (g 8)` returns 42 – `(g 8)` is never evaluated, and the program terminates normally. You now have a concrete example that illustrates why understanding the computation model is important.

Our study of computation models shall focus on the recognition and the generation of language elements. A *language* is a set of *words* over a given alphabet. A word is an ordered collection of alphabet elements, read left to right, that may contain repetitions. The number of words in a language may be finite or infinite. For example, the following defines the language containing all words of length less than or equal to 2 over the alphabet `(a b)`:

`LT4 =` $\{\epsilon$ `a b aa ab ba bb`$\}$

Given that the language is finite, it suffices to list its elements. We denote the empty word (the word with zero alphabet elements) as ϵ. This word has length zero. In contrast, `bb` has length 2.

The following defines the infinite language for all strings that end with an `a` over `(a b)`:

`ENDA =` $\{$`w | w ends with an a`$\}$

In this definition, a `set former` is used. This is necessary because it is impossible to list all the elements of the language. You may read the definition as stating that the language `ENDA` is equal to the set of words made from zero or more `a`s and `b`s that end with an `a`.

The computation needed to determine if a word, w, is a member of a language L is performed by a *finite-state machine* (aka a finite-state automaton). These theoretical machines have led to efficient algorithms to determine if a pattern is found in a strand of DNA or in a block of text. The computations needed to generate a word w that is a member of a language L are done using a grammar. The study of grammars has led to the development of parsers, interpreters, and compilers for programming languages and to natural language processing. You should dispel any misconception that the study of automata theory and formal languages has no practical applications relevant to the life of a problem-solver and programmer.

11 Essential Mathematical Background

11.1 Sets

A set is a collection of items without repetitions. These are used, for example, to represent alphabets and languages. For instance, an alphabet, Σ, may be defined to be a set of alphabetic characters, and a language may be defined as a set of words formed by the characters in Σ:

$$\Sigma = \{a\ b\ c\}$$

$$L = \{w \mid w \in \Sigma^* \wedge w \text{ has an even length}\}$$

The above definitions state that Σ is the alphabet containing a, b, and c and that L is the set of words formed using zero or more elements of Σ that have an even length. The *, called Kleene star, stands for zero or more elements of the set. A $^+$ stands for one or more elements of the set. Throughout this book, Σ is used to represent an alphabet. The symbol $\wedge$ means and. It is opportune at this time to note that the symbol $\vee$ means or.

11.1.1 Set Notation

Finite sets are specified by listing their elements inside curly braces like in the definition above for. Infinite sets are represented using a set former. A set former specifies the conditions that must hold in order for an element to be a member of a set. In general, set formers are written as follows:

$$\{w \mid P(w)\}$$

This means that w is a member of the set if the predicate P holds. In the definition for L above, we have that:

$$P(w) = w \in \Sigma^* \wedge w\text{'s length is even}$$

1 Let $\Sigma = \{a\ b\}$. Define the set of all words of length less than or equal to 5.

2 Let $\Sigma = \{a\ b\}$. Define the set of all words that contain an even number of bs.

3 Define the set of all even integers. Make sure to define the alphabet for this language.

If every element of A is a member of B, then A is a *subset* of B. We denote this as:

$$A \subseteq B$$

If every element of A is a member of B and not all elements of B are in A, then A is a proper subset of B. We denote this as:

$$A \subset B$$

For example, consider the following sets for Σ = (a b):

STA = {w | w starts with an a}
SEA = {w | w starts or ends with an a}

We have that STA $\subset$ SEA because:

- Every word that starts with an a is also a word that starts or ends with an a
- There are words that start with a b and end with an a in SEA

We say that two sets are equal, A = B, if and only if A $\subseteq$ B and B $\subseteq$ A. That is, every element of A is a member of B, and every element of B is a member of A.

11.2 Set Operations

Sets may be combined to create new sets. This is done through set operations. You may think of a set operation as a function on one or more sets. There are six basic set operations: union, intersection, difference, complement, cross product, and power set. For the discussion below, assume that Σ = (a b).

The union of two sets, A and B, is a set that contains all the elements of A and all the elements of B. We denote the union of two sets as follows:

$$A \cup B = \{x |\ x \in A \vee x \in B\}$$

For instance, consider the following sets:

$$\texttt{A} = \{\texttt{w} \mid \texttt{w starts with an a}\}$$
$$\texttt{B} = \{\texttt{w} \mid \texttt{w starts with an b}\}$$

We have that:

$$\texttt{A} \cup \texttt{B} = \{\texttt{w} \mid \texttt{w starts with an a} \vee \texttt{w starts with an b}\}$$

4 Is $\texttt{A} \cup \texttt{B} = \Sigma^*$? Why or why not?

The intersection of two sets, A and B, is a set that contains all the elements that A and B have in common. We denote the intersection of two sets as follows:

$$\texttt{A} \cap \texttt{B} = \{\texttt{x} \mid \texttt{x} \in \texttt{A} \wedge \texttt{x} \in \texttt{B}\}$$

For instance, once again, consider the following sets:

$$\texttt{A} = \{\texttt{w} \mid \texttt{w starts with an a}\}$$
$$\texttt{B} = \{\texttt{w} \mid \texttt{w starts with an b}\}$$

We have that:

$$\begin{aligned}\texttt{A} \cap \texttt{B} &= \{\texttt{w} \mid \texttt{w starts with an a} \wedge \texttt{w starts with an b}\} \\ &= \emptyset\end{aligned}$$

The intersection of A and B is the empty set because there are no words that start with both an a and a b. We say that two sets whose intersection is empty are *mutually exclusive*. In contrast, consider the following two sets:

$$\texttt{C} = \{\epsilon\ \texttt{a aa aaa aaaa aaaaa aaaaaa}\}$$
$$\texttt{D} = \{\texttt{w} \mid \texttt{w} \in \texttt{a}^* \wedge \texttt{w has an even number of a}\}$$

The intersection of these two sets is:

$$\texttt{C} \cap \texttt{D} = \{\epsilon\ \texttt{aa aaaa aaaaaa}\}$$

It contains only the words that have an even number of as that are members of both sets.

The difference of two sets, A and B, is the set that contains all the elements of A that are not in B:

$$\texttt{A} - \texttt{B} = \{\texttt{w} \mid \texttt{w} \in \texttt{A} \wedge \texttt{w} \notin \texttt{B}\}$$

Once again, consider the sets C and D defined above. We have that:

$$D - C = \{w \mid w \in a^* \ \wedge \ w \text{ has an even number of a} \wedge |w| \geq 8\}$$

$$C - D = \emptyset$$

Here, |w| denotes the length of w. If all the strings of as of length less than or equal to 6 are removed from D, then all the strings of as with a length greater than or equal to 8 remain. C - D is the empty set because all the words in C are in D.

The complement of a set, A, is denoted $\bar{A}$. It contains all possible words built using the elements of Σ that are not in A. We may specify it as follows:

$$\bar{A} = \Sigma^* - A$$

Σ^* is the language of all words that may be built using the elements of Σ. The difference of this language and A is the language containing all the strings not in A. For instance, consider the following language:

$$M = \{w \mid w \text{ has exactly two b}\}$$

The complement of M is:

$$\bar{M} = \Sigma^* - M = \{w \mid \text{the number of b in w is not two}\}$$

The cross product of two languages, A and B, is the set of all pairs such that the first element is a member of A and the second element is a member of B:

$$A \times B = \{(a\ b) \mid a \in a \wedge b \in B\}$$

Consider the following sets:

$$E = \{x\ y\ z\}$$

$$F = \{0\ 1\}$$

$$\mathbb{N} = \{n \mid n \text{ is a natural number}\}$$

The symbol $\mathbb{N}$ denotes the set of natural numbers, that is, the set of integers greater than or equal to zero. We have that:

$$E \times F = \{(x\ 0)\ (y\ 0)\ (z\ 0)\ (x\ 1)\ (y\ 1)\ (z\ 1)\}$$

What is $\mathbb{N} \times \mathbb{N}$? It is impossible to write out the elements of the set because the set is infinite. Therefore, we must use a set former to define it. If you think about it carefully, the pairs in this set are the integer points in the first quadrant of a two-dimensional Cartesian plane including the points on the vertical and horizontal axes. Therefore, we may write:

$$\mathbb{N} \times \mathbb{N} = \{(x\ y) \mid x,y \in \mathbb{N} \wedge x \geq 0 \wedge y \geq 0)\}$$

Observe that when we write a definition, we must be very precise and leave no room for ambiguity. It would be incorrect to write:

$$\mathbb{N} \times \mathbb{N} = \{(x\ y) \mid x \geq 0 \wedge y \geq 0)\}$$

This is incorrect because it does not precisely specify that `x` and `y` must be natural numbers.

The power set of a set `A`, 2^{A}, is the complete set of subsets of `A`, that is, the union of the subsets that have zero elements of `A`, that have one element of `A`, that have two elements of `A`, and so on until the subset that has `n` elements of `A` where `n = |A|` (the number of elements in `A`). Consider the following languages:

$$\texttt{EMPTY} = \{\}$$

$$\texttt{SET1} = \{\texttt{r e a}\}$$

We can observe that the power set of each is:

$$2^{\texttt{EMPTY}} = \{\{\}\}$$

$$2^{\texttt{SET1}} = \{\{\texttt{r e a}\}\ \{\texttt{r e}\}\ \{\texttt{r a}\}\ \{\texttt{r}\}\ \{\texttt{e a}\}\ \{\texttt{e}\}\ \{\texttt{a}\}\ \{\}\}$$

The power set of `EMPTY` only contains itself because the empty set has no elements. The power set of `SET1` contains all the subsets of sizes 0–3. It is difficult to write a set form for 2^{A}. The good news is that there is an alternative way to specify a set. A constructor may be written. That is, an algorithm may be implemented to build the power set of `A`. As a computer scientist or programmer, this approach is likely to appeal to you. Consider an arbitrary $\texttt{a} \in \texttt{A}$. An arbitrary element of 2^{A} either contains or does not contain `a`. This suggests an algorithm to compute 2^{A}. If `A = {}`, then the power set is $\{\{\}\}$. If $\texttt{A} \neq \{\}$, compute the power set of `A - {a}`. Call this set `P`. The power set is obtained by the union of `P` and the set obtained from adding `a` to every set in `P`. Following the steps of the design recipe yields the program[3] displayed in Fig. 4.

11.2.1 Set Laws

Laws may be thought of provable rules that reveal properties of sets. They may be used to prove further properties (i.e., theorems) about sets. The laws are usually organized in six categories: idempotency, commutativity, associativity, distributivity, absorption, and De Morgan's laws.

The idempotent laws state that the union and the intersection of a set, `A`, with itself are `A`:

$$\texttt{A} \cup \texttt{A} = \texttt{A}$$

$$\texttt{A} \cap \texttt{A} = \texttt{A}$$

[3] `#|` and `|#` are used to write multiline comments.

Fig. 4 A function to compute the power set of a set

```
#lang fsm

#|
 Data Definitions
   A list of X, lox, is either:
     1. '()
     2. (cons X lox)

   A set of X, setx, is a (listof X)
|#

;; Sample setx
(define EMPTY-SET '())
(define SET1 '(r e a))

;; setx → (listof setx)
;; Purpose: Return the power set of the given set
(define (powerSet A)
  (cond [(null? A) (list '())]
        [else
         (let ((rest (powerSet (cdr A))))
           (append
            (map (lambda (x) (cons (car A) x)) rest)
            rest))]))

(check-equal? (powerSet EMPTY-SET) '(()))
(check-equal? (powerSet SET1)
              '((r e a) (r e) (r a) (r) (e a) (e) (a) ()))
```

Let us prove the first.

Theorem 1 $A \cup A = A$

Proof We shall prove that:
(a) A ∪ A ⊆ A
(b) A ⊆ A ∪ A

(a) Assume x is an arbitrary element in A ∪ A. This means that x ∈ A or x ∈ A. Hence, x ∈ A. Therefore, we may conclude that A ∪ A ⊆ A.

(b) Assume that x is an arbitrary element in A. This means that x ∈ A ∪ A. Therefore, we may conclude that A ⊆ A ∪ A.

Therefore, the following implication holds:
A ∪ A ⊆ A ∧ A ⊆ A ∪ A ⇒ A = A ∪ A. □

5 Prove that `A ∩ A = A`.

The commutative laws are:

```
A ∪ B = B ∪ A

A ∩ B = B ∩ A
```

Let us prove the second.

Theorem 2 $A \cap B = B \cap A$

Proof We shall prove that:
(a) A ∩ B ⊆ B ∩ A
(b) B ∩ A ⊆ A ∩ B

(a) Assume x is an arbitrary element in A ∩ B. This means that x ∈ A and x ∈ B. Hence, x ∈ B ∩ A. Thus, we may conclude that A ∩ B ⊆ B ∩ A.

(b) Assume that x is an arbitrary element in B ∩ A. This means that x ∈ B and x ∈ A. This means that x ∈ B and x ∈ A. Hence, x ∈ A ∩ B. Thus, we may conclude that B ∩ A ⊆ A ∩ B.

Therefore, the following implication holds:
A ∩ B ⊆ B ∩ A ∧ B ∩ A ⊆ A ∩ B ⇒ A ∩ B = B ∩ A. □

6 Prove that `A ∪ B = B ∪ A`.

The associative laws are:

```
(A ∪ B) ∪ C = A ∪ (B ∪ C)

(A ∩ B) ∩ C = A ∩ (B ∩ C)
```

Let us prove the first.

Theorem 3 $(A \cup B) \cup C = A \cup (B \cup C)$

Proof We shall prove that:
(a) (A ∪ B) ∪ C ⊆ A ∪ (B ∪ C)
(b) A ∪ (B ∪ C) ⊆ (A ∪ B) ∪ C

(a) Assume `x` is an arbitrary element of (A ∪ B) ∪ C. This means that:

```
   x ∈ (A ∪ B) ∨ x ∈ C
⇒ x ∈ A ∨ x ∈ B ∨ x ∈ C
⇒ x ∈ A ∪ (B ∪ C)
```

Thus, we may conclude that `(A ∪ B) ∪ C ⊆ A ∪ (B ∪ C)`.

(b) Assume x is an arbitrary element of A ∪ (B ∪ C). This means that:

$$\begin{aligned} & x \in A \vee x \in (B \cup C) \\ \Rightarrow\ & x \in A \vee x \in B \vee x \in C \\ \Rightarrow\ & x \in (A \cup B) \cup C \end{aligned}$$

Thus, we may conclude that $A \cup (B \cup C) \subseteq (A \cup B) \cup C$.

Therefore, the following implication holds:

$$\begin{aligned} & (A \cup B) \cup C \subseteq A \cup (B \cup C) \\ & \wedge\ A \cup (B \cup C) \subseteq (A \cup B) \cup C \\ \Rightarrow\ & \\ & (A \cup B) \cup C = A \cup (B \cup C). \end{aligned}$$

□

7 Prove that $(A \cap B) \cup C = (A \cup C) \cap ((B \cup C)$.

The absorption laws are:

$$(A \cup B) \cap A = A$$

$$(A \cap B) \cup A = A$$

Let us prove the first.

Theorem 4 *(A ∪ B) ∩ A = A*

Proof We shall prove that:
(a) (A ∪ B) ∩ A ⊆ A
(b) A ⊆ (A ∪ B) ∩ A

(a) Assume x is an arbitrary element of (A ∪ B) ∩ A. This means that:

$$x \in (A \cup B) \wedge x \in A$$

Therefore, we may conclude that $(A \cup B) \cap A \subseteq A$.

(b) Assume x is an arbitrary element of A. This means that:

$$x \in (A \cup B) \Rightarrow x \in (A \cup B) \cap A$$

Therefore, we may conclude that $A \subseteq (A \cup B) \cap A$.

Thus, the following implication holds:

$$\begin{aligned} & (A \cup B) \cap A \subseteq A \wedge A \subseteq (A \cup B) \cap A \\ \Rightarrow\ & \\ & (A \cup B) \cap A = A. \end{aligned}$$

□

8 Prove that $A \cap B) \cup A = A$.

De Morgan's laws are:

$$A - (B \cup C) = (A - B) \cap (A - C)$$

$$A - (B \cap C) = (A - B) \cup (A - C)$$

Let us prove the first.

Theorem 5 *A - (B ∪ C) = (A - B) ∩ (A - C)*

Proof We shall prove that:
(a) $A - (B \cup C) \subseteq (A - B) \cap (A - C)$
(b) $(A - B) \cap (A - C) \subseteq A - (B \cup C)$

(a) Assume x is an arbitrary element of $A - (B \cup C)$. This means that:

$$\begin{aligned} & x \in A \wedge x \notin B \wedge x \notin C \\ \Rightarrow\ & x \in (A - B) \wedge x \in (A - C) \end{aligned}$$

Therefore, we may conclude that $A - (B \cup C) \subseteq (A - B) \cap (A - C)$.

(b) Assume x is an arbitrary element of $(A - B) \cap (A - C)$. This means that:

$$\begin{aligned} & x \in (A - B) \wedge x \in (A - C) \\ \Rightarrow\ & x \in A \wedge x \notin B \wedge x \notin C \\ \Rightarrow\ & x \notin (B \cup C) \\ \Rightarrow\ & x \in A - (B \cup C) \end{aligned}$$

Therefore, we may conclude that $(A - B) \cap (A - C) \subseteq A - (B \cup C)$.

Thus, the following implication holds:

$$\begin{aligned} & A - (B \cup C) \subseteq (A - B) \cap (A - C) \\ \wedge\ & (A - B) \cap (A - C) \subseteq A - (B \cup C) \\ \Rightarrow\ & \\ & A - (B \cup C) = (A - B) \cap (A - C). \end{aligned}$$

□

9 Prove that $A - (B \cup C) = (A - B) \cap (A - C)$.

11.3 Relations and Functions

Relations associate an element of the domain (the input set) with an element of the range (the output set). For instance, the binary relational operator $\geq$ has as its domain pairs (or tuples) of real numbers and has as its range the Booleans (i.e., true and false). Relations define a language (or if you like a set). The members of such a language are the elements that are related. The language defined by $\geq$ may be specified as follows:

$$\texttt{GEQ} = \{\texttt{(x y) | x, y} \in \mathbb{R} \wedge \texttt{x} \geq \texttt{y}\}$$

The above states that `x` and `y` are real numbers and that `x` is greater than or equal to `y`. `(31 7)` and `(8 4)`, for example, are members of `GEQ`. `(21 55)` and `(-8 40)` are not members of `GEQ`. If `(a b)` is a member of a relation, then we say that `a` and `b` are related and it is denoted by `aRb`.

A special type of relation is called a `function`. A function is a binary relation that associates a member from its domain with a unique member from its range. More formally, a function, `f`, from a set `A` to a set `B`, defines a language of ordered pairs, `(a b)`, such that there is exactly one ordered pair for every element of `A`. For instance, consider the following sets:

$$\texttt{NAMES} = \{\texttt{w | w is a first name}\}$$

$$\texttt{PASSP} = \{\texttt{p | p is a person with a single passport}\}$$

The relation that maps a person with a single passport to a first name is specified as follows:

$$\texttt{R}_1 = \{\texttt{(p n) | p} \in \texttt{PASSP} \wedge \texttt{n} \in \texttt{NAMES}\}$$

$\texttt{R}_1$ is a function because every single passport holder has a first name. On the other hand, consider this relation:

$$\texttt{R}_2 = \{\texttt{(n p) | p} \in \texttt{PASSP} \wedge \texttt{n} \in \texttt{NAMES}\}$$

$\texttt{R}_2$ is not a function because more than one person with a single passport may have the same first name. Take note that the elements of a relation are not sets. They are ordered pairs. That is, `(p n)` is not the same `(n p)`.

We say that a function, `f`, is *one-to-one* if for any distinct `a` and `b`, `f(a)` $\neq$ `f(b)`. For instance, consider the following sets:

$$\texttt{STATES} = \{\texttt{s | s is a state in the USA}\}$$

$$\texttt{STCAPS} = \{\texttt{c | c is a state capital in the USA}\}$$

The following is a one-to-one function:

$$\texttt{R}_3 = \{\texttt{(c s) | c} \in \texttt{STCAPS} \wedge \texttt{s} \in \texttt{STATES}\}$$

This is a one-to-one function because every state capital is the capital of a different state. We say that a function, `f`, is *onto* if each element in the range is mapped to by at least one element in the domain. R_3 is onto because every state has a state capital. Finally, a function is a bijection if it is both one-to-one and onto. R_3 is a bijection because it is both one-to-one and onto.

11.4 Countable and Uncountable

As you can imagine, the study of formal languages involves infinite sets. We must be careful about how we reason about infinite sets. At an intuitive level, an infinite set is different from a finite set because all the members of the infinite set cannot be listed in a finite amount of time. If you understand this observation, then it is easy to see that the size or *cardinality* of a finite set is less than the cardinality of an infinite set.

Two sets, `A` and `B`, have the same cardinality, `|A|` and `|B|`, if there is a bijection between `A` and `B`. The existence of such a bijection informs us that the sets are *equinumerous*. The concept of equinumerous is slippery when it comes to infinite sets. Intuitively, it is easy to believe that the cardinality of two infinite sets is the same. After all, both sets have a never-ending number of elements. That is, they have the same size. Does this mean that all infinite sets are equinumerous? It turns out that the answer is no. In other words, there exist infinite sets, `|C|` and `|D|`, such that a bijection between them does not exist. To intuitively understand this, consider the set of real numbers, $\mathbb{R}$, and, $\mathbb{Z}$, the set of integers. Both sets are infinite, but they are not equinumerous. That is, they do not have the same cardinality. How can we prove this? It is tempting to say that $\mathbb{Z} \subset \mathbb{R}$ and, therefore, there are more real numbers than integers. Thus, $\mathbb{Z}$ and $\mathbb{R}$ do not have the same cardinality and are not equinumerous. This, however, is fallacious reasoning. It says nothing about whether or not there exists a bijection between $\mathbb{Z}$ and $\mathbb{R}$. We need to argue that a bijection does not exist. We shall hold off on this argument until we learn about diagonalization proofs in the next chapter.

We can now formally define a finite set. A set, `A`, is finite if it is equinumerous with the first `n` elements of $\mathbb{N}$. We say that `n` is the cardinality of `A`. For example, consider:

```
A = {f o r {m a l}}
```

There is a bijection, `g`, between `A` and {`0 1 2 3`}:

```
g(0) = f      g(1) = o
g(2) = r      g(3) = {m a l}
```

The `|A|` is 4, and `A` is, therefore, finite.

A set is infinite if it is not finite. The sets $\mathbb{R}$, $\mathbb{N}$, and $\mathbb{Z}$ are infinite. There is no bijection between the elements of these sets and the first `n` elements of

$\mathbb{N}$. Are any of these sets equinumerous? $\mathbb{N}$ and $\mathbb{Z}$ are equinumerous. Observe that despite $\mathbb{N} \subset \mathbb{Z}$, these sets are equinumerous. The following is a bijection, $\mathbb{N} \rightarrow \mathbb{Z}$, between these two sets:

$$f(n) = \begin{cases} \frac{n}{2} & \texttt{if n is even} \\ -\frac{n+1}{2} & \texttt{if n is odd} \end{cases}$$

10 Consider the following sets:

```
S1 = {x | x is an even integer}
S2 = {x | x is an integer}
```

Do they have the same cardinality?

11 Consider the following sets:

```
S1 = {x | x is the square of an integer}
S2 = {x | x is the cube of an integer}
```

Do they have the same cardinality?

A set is *countably infinite* if it is equinumerous with $\mathbb{N}$. Intuitively, this means that a program may be written to print out the members of the set. The program, of course, will run forever, but if you wait long enough, any arbitrary element of the set will eventually be printed.

$\mathbb{N}$ is countably infinite. The `FSM` program to print the natural numbers is displayed in Fig. 5. The function `print-natnums` takes no arguments and returns `(void)` (because it is only called for the effect of printing to the screen). It calls a local auxiliary function, `printer`, with 0 to print the natural numbers starting at 0. This function traverses the natural numbers starting with the given natural number. If the given natural number is positive infinity (i.e., `+inf.0`), the function halts and returns `(void)`. Otherwise, it prints the given natural number and recursively processes the next natural number. Given enough time and memory, a call to `print-natnums`, `(print-natnums)`, runs forever, but eventually it will print any arbitrary natural number.

A set is *countable* if it is finite or it is countably infinite. A set is *uncountable* if it is not countable. Intuitively, a set is uncountable if a program that eventually prints any arbitrary element cannot be written. To demonstrate that a set is countable, it suffices to write a program to print its elements guaranteeing that eventually any arbitrary element is eventually printed. Sometimes, this requires careful design and creativity. For instance, consider the following sets:

Fig. 5 A program to print the natural numbers

```
#lang fsm

;;  → (void)
;; Purpose: Print the natural numbers
(define (print-natnums)
  ;; natnum → (void)
  ;; Purpose: Print the natural numbers starting with
  ;;          the given natural number
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln n)
          (printer (add1 n)))))
  (printer 0))
```

$$\texttt{EVENNATS} = \{\texttt{2n} \mid \texttt{n} \in \mathbb{N}\}$$
$$\texttt{MULTSOF3} = \{\texttt{3n} \mid \texttt{n} \in \mathbb{N}\}$$
$$\texttt{A}^* = \{\texttt{w} \mid \texttt{w} \in \texttt{a}^*\}$$
$$\texttt{BIGSET} = \texttt{EVENNATS} \cup \texttt{MULTSOF3} \cup \texttt{A}^*$$

Is BIGSET countable? We can observe that EVENNATS, MULTSOF3, and $\texttt{A}^*$ are countable. The proof is displayed in Fig. 6.[4] To prove that BIGSET is countable, we need a program to print its elements. A possible design prints the even natural numbers, then the multiples of 3, and then the elements of $\texttt{a}^*$ as follows:

```
;;  → (void)
;; Purpose: Print the elements of BIGSET
(define (print-bigset)
  (begin
    (print-even-natnums)
    (print-mults3)
    (print-a*)))
```

Unfortunately, this design does not work. Only the elements of the even natural numbers are ever printed. The multiples of 3 and the elements of $\texttt{A}^*$ are never printed. Therefore, the above program fails to prove that BIGSET is countable. We need to return to the design table or argue that BIGSET is uncountable.

A dovetailing strategy yields a different design. The idea is to smoothly fit together printing members of all three sets before moving forward with the

[4] The evaluation of (build-list n (λ (i) #\a)) produces a list with `n` `a` characters. The function `list->string` converts a list of characters into a string.

Fig. 6 Printing even natural numbers, multiples of 3, and $\texttt{a}^*$ elements

```
#lang fsm

;;  → (void)
;; Purpose: Print the even natural numbers
(define (print-even-natnums)
  (define (printer n)
    ;; natnum → (void)
    ;; Purpose: Print the multiples of 2 starting with the
    ;;          multiple of 2 for the given natural number
    (if (= n +inf.0)
        (void)
        (begin
          (displayln (* 2 n))
          (printer (add1 n)))))
  (printer 0))

;;  → (void)
;; Purpose: Print the multiples of 3
(define (print-mults3)
  ;; natnum → (void)
  ;; Purpose: Print the multiples of 3 starting with the
  ;;          multiple of 3 for the given natural number
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln (* 3 n))
          (printer (add1 n)))))
  (printer 0))

;;  → (void)
;; Purpose: Print the elements of a*
(define (print-a*)
  ;; natnum → (void)
  ;; Purpose: Print words of a's starting with the word
  ;;          with the given number of a's
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln (list->string (build-list n (λ (i) #\a))))
          (printer (add1 n)))))
  (printer 0))
```

next element of any set. Recall that all three sets are countable. This means that each set is equinumerous with $\mathbb{N}$ and, therefore, there is a bijection from the natural numbers to the member of each set. This suggests the program can print an element of each set as it traverses the natural numbers. Given a natural number, `n`, print the $\texttt{n}^{\texttt{th}}$ element of `EVENNATS`, of `MULTS3`, and of $\texttt{A}^*$,

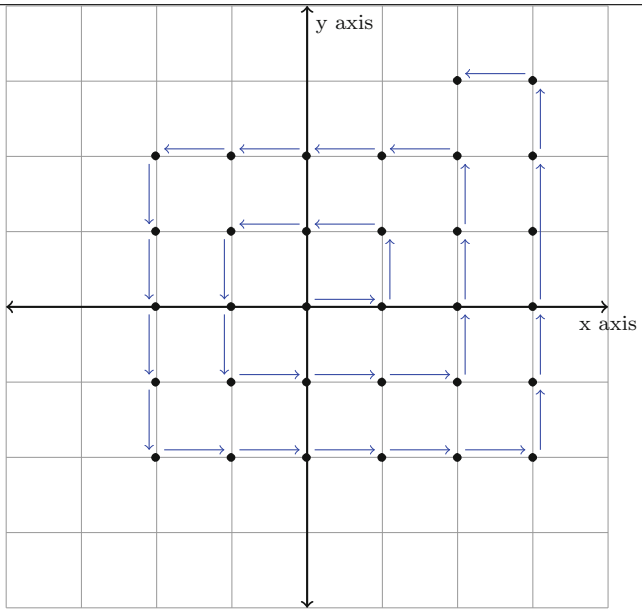
and then continue traversing the natural numbers. The result of this design is:

```
;;  → (void)
;; Purpose: Print the elements of BIGSET
(define (print-bigset)
  ;; natnum → (void)
  ;; Purpose: Print the elements of EVENNATS, MULTS3, and
  ;;          A* starting with the elements indexed by
  ;;          the given natural number
  (define (printer n)
    (if (= n +inf.0)
        (void)
        (begin
          (displayln (* 2 n))
          (displayln (* 3 n))
          (displayln
            (list->string (build-list n (λ (i) #\a))))
          (printer (add1 n)))))
  (printer 0))
```

It is clear that eventually any arbitrary element of `BIGSET` is printed by the above program. Thus, we may conclude the `BIGSET` is countable.

Dovetailing is a powerful design technique. Consider the set of integer points, `(x, y)`, in a two-dimensional Cartesian plane. Is this set countable? We can observe that there are four quadrants. An initial attempt to develop an algorithm to print all the integer points may suggest printing the points in the first quadrant, then in the second quadrant, then in the third quadrant, and finally in the fourth quadrant. Unfortunately, such a design fails because the points in each quadrant are infinite. Therefore, if possible, the points in the first quadrant are printed, but none of the points in other quadrants are ever printed. If this set is countable, we need a different design. The set of integer points is infinite, and you ought to consider dovetailing. We need to progressively print the points in each of the four quadrants. How can this be done? We can think of the Cartesian plane consisting of incrementally bigger squares outlined by integer points defined by their distance from the origin. The first square contains the points that vary either on the `x` or on the `y` only by 0 (i.e., only the origin). The second square contains the points that vary either on the `x` or on the `y` only by 1. The third square contains the points that vary either on the `x` or on the `y` only by 2. The process continues in this manner until the distance used to define the square reaches positive infinity. How can the points be printed using this design? Figure 7 displays an ordering to print all the integer points. The algorithm starts by printing the origin. The blue arrows illustrate the order in which to print the points. Following the blue arrows dovetails around all four quadrants printing the integer points that outline increasingly lager squares. It is not difficult to

Fig. 7 Dovetailing to print the integer pairs in a 2D Cartesian plane



see that eventually any arbitrary integer point is printed. In essence, the integer points of a square in all four quadrants are printed before moving onto increasing the length of the square by 1. This algorithm proves that the set of integer points is countable.

12 Prove that the set of integer points in the first quadrant is countable.

13 Prove that the set of finite subsets of $\mathbb{N}$ is countable.

14 Prove that A = {1 2 3 4 5} is countable.

15 Consider the following alphabet:

Σ = {a b}

Prove that Σ^* is countable.

16 Prove that the union of a countably infinite number of countably infinite sets is countable.

17 Design and implement a function to print the integer pairs in a 2D Cartesian plane.

Chapter 3
Types of Proofs

A proof is an argument that establishes that a hypothesis is true. It uses assumptions to reach a conclusion. Developing a proof can be challenging. There are, however, some fundamental proof techniques that can help guide the process. There are formal logic proofs, mathematical induction proofs, pigeonhole principle proofs, proofs by contradiction, and diagonalization proofs. This is not an exhaustive list of the types of proofs that exist, but it is the list of proof types you need to be aware of to successfully navigate this textbook. Do not worry if you are not an expert at writing proofs. You shall learn as you progress through this textbook.

When developing a proof, it is important to precisely state the assumptions and state the conclusion (the statement you want to prove). Failing to do so is tantamount to writing a function without knowing its inputs and its purpose. Writing precise statements takes practice, and you ought to be patient with yourself as you learn.

12 Formal Logic Proofs

Logic may be defined as the study of (correct) reasoning. Formal logic is based on inferences that can be proven to always be true. For the purposes of this textbook, we are mostly interested in proving conjunctions, disjunctions, implications, and equivalences.

A conjunction that is true means that all input statements are true. It is denoted using $\wedge$ between the statements. For example, for two statements, `A` and `B`, their conjunction is `A` $\wedge$ `B`. It is read as `A` and `B`. If we know that `A` and `B` are true, then we may conclude that `A` $\wedge$ `B` is true. An arbitrary statement, of course, may or may not be true. What is the value of a conjunction when

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_3

one of its inputs is false? The following is the truth table for $\wedge$ with two input statements:

A	B	A $\wedge$ B
False	False	False
False	True	False
True	False	False
True	True	True

The table informs us that a conjunction is true only when all its input statements are true. Otherwise, it is false. This means that if we wish to prove a conjunction true, we must demonstrate that the input statements are true. For instance, consider proving the following conjunction:

```
28 + 2 + 1 = 31 ∧ 2 * 3 + 4 = 10
```

We must prove that `28 + 2 + 1 = 31` is true and that `2 * 3 + 2 = 10` is true. To prove an equality is true, we can make both sides of the equality the same without moving elements from one side of the `=` to the other side.[5] To achieve this, we can work with the left-hand sides as follows:

```
28 + 2 + 1 = 31 ∧ 2 * 3 + 4 = 10
    30 + 1 = 31 ∧     6 + 4 = 10
        31 = 31 ∧        10 = 10
```

Clearly, `31 = 31` is true and `10 = 10` is true. Thus, we may conclude that the conjunction above holds (i.e., it is true).

1 Prove the following:

1. $3 * 5 + 7 * 8 \neq 100 \wedge \frac{60 + 6}{3} = 22$
2. $A \wedge A = A$
3. $A \wedge (9 * 9 = 91) = \text{False}$

A disjunction that is true means that any of its input statements are true. It is denoted using $\vee$ between the statements. For example, for two statements, `A` and `B`, their disjunction is `A ∨ B`. It is read as `A` or `B`. The following is the truth table for $\vee$ with two input statements:

A	B	A $\vee$ B
False	False	False
False	True	True
True	False	True
True	True	True

[5] Remember that the rules of algebra are only valid if we know that both sides of the equality are the same value.

The table informs us that a disjunction is true if any of its input statements are true. Otherwise, it is false. This means that if we wish to prove a disjunction true, we must demonstrate that at least one of its input statements is true. For instance, consider proving the following disjunction:

$$\texttt{7 + 1 + 3 > 20} \quad \vee \quad \texttt{-2 * 2 + 4} \geq \texttt{0} \quad \vee \quad \texttt{x} \geq \texttt{x + x + x}$$

We must prove that one of the statements is true. We can simplify the inequalities as follows:

$$\begin{aligned} \texttt{7 + 1 + 3 > 20} \quad &\vee \quad \texttt{-2 * 2 + 4} \geq \texttt{0} \quad \vee \quad \texttt{x} \geq \texttt{x + x + x} \\ \texttt{8 + 3 > 20} \quad &\vee \quad \texttt{-4 + 4} \geq \texttt{0} \quad \vee \quad \texttt{x} \geq \texttt{x + 2x} \\ \texttt{11 > 20} \quad &\vee \quad \texttt{0} \geq \texttt{0} \quad \vee \quad \texttt{x} \geq \texttt{3x} \end{aligned}$$

Clearly, $\texttt{0} \geq \texttt{0}$. Therefore, we may conclude that the disjunction holds.

2 Prove the following:

1. $3 * 5 + 7 * 8 \neq 100 \vee \frac{60 + 6}{3} = 22$
2. $A \vee (A \wedge A) = A$
3. $A \vee (9 * 9 = 81) = \text{True}$

An important logical statement that we must know how to prove is an implication: $\texttt{A} \Rightarrow \texttt{B}$. It is read as `A` implies `B`. `A` is called the antecedent, and `B` is called the consequent. It states that if `A` is true, then `B` is also true. The truth table for implication is:

A	B	A ⇒ B
False	False	True
False	True	True
True	False	False
True	True	True

This table informs us that to prove that an implication is true, we must prove that when the antecedent is true, then the consequent is also true. If this is achieved, then the third line of the truth table does not occur, and the implication is always true. Observe that if the antecedent is false, then the value of the consequent does not matter because the implication is always true. Therefore, to prove an implication holds, we assume the antecedent is true and demonstrate that the consequent is true. For instance, consider proving the following implication:

$$\texttt{3x - 9} \geq \texttt{0} \Rightarrow \texttt{x} \geq \texttt{3}$$

Assume that $3x - 9 \geq 0$ is true. We must show that $x \geq 3$ is true. Given that we have assumed that the antecedent is true, we are free to manipulate it (e.g., using algebra):

$$\begin{aligned} 3x - 9 &\geq 0 \Rightarrow x \geq 3 \\ 3x &\geq 9 \Rightarrow x \geq 3 \\ x &\geq 3 \Rightarrow x \geq 3 \end{aligned}$$

Clearly, if $x \geq 3$ is true on the left-hand side of the implication, it is also true on the right-hand side of the implication. Therefore, we may conclude that the implication holds.

3 Prove that x is a multiple of 31 greater than $0 \Rightarrow x > 10$.

4 Prove that $A \subseteq B \Rightarrow A \cap C \subseteq B$.

5 Prove that if $|x - 5| > 0$, then $x > 5 \vee x < -5$.

6 Prove that $3 = 4 \Rightarrow 10 = 10$.

7 Prove that $x < y \wedge y < z \Rightarrow x < z$.

Another important logical statement that we must know how to prove is logical equivalence: $A \Leftrightarrow B$. It is read as A is equivalent to B. It is shorthand notation for the conjunction of two implications:

$$A \Rightarrow B \wedge B \Rightarrow A$$

The truth table for logical equivalence is:

A	B	A ⇔ B
False	False	True
False	True	False
True	False	False
True	True	True

A logical equivalence holds when both input statements have the same value. To prove that an equivalence holds, we must eliminate the possibility that lines 2 and 3 in the table above can occur with the given statements. Observe that if $A \Rightarrow B$ holds, then the input statements cannot satisfy line 3 in the table above. Similarly, if $B \Rightarrow A$ holds, then the input statements cannot satisfy line 2 in the table above. Therefore, the table informs us (as you might have suspected) proving logical equivalence requires proving two implications. Each requires an independent and separate proof.

For instance, consider proving the following equivalence for a natural number x:

$$x \text{ is a square} \Leftrightarrow \sqrt{x} \in \mathbb{N}$$

We must prove that:

1. `x is a square` $\Rightarrow \sqrt{x} \in \mathbb{N}$
2. $\sqrt{x} \in \mathbb{N} \Rightarrow$ `x is a square`

To prove the first implication, we assume `x is a square` and show that $\sqrt{x} \in \mathbb{N}$:

$$\begin{aligned} \texttt{x is a square} &\Rightarrow \texttt{x = k * k, k} \in \mathbb{N} \\ &\Rightarrow \sqrt{x} = \texttt{k} \\ &\Rightarrow \sqrt{x} \in \mathbb{N} \end{aligned}$$

The above chain of implications proves that the first implication holds. For the second implication, assume $\sqrt{x} \in \mathbb{N}$ is true, and show that `x is a square`:

$$\begin{aligned} \sqrt{x} \in \mathbb{N} &\Rightarrow \texttt{x = k * k} \\ &\Rightarrow \texttt{x = k}^2 \end{aligned}$$

The above chain of implications establishes that `x` is a square. Given that we have established both implications, we may conclude that the equivalence holds.

8 Prove that A $\Rightarrow$ B $\Leftrightarrow \neg$A $\vee$ B, where $\neg$A is the negation of A (if A is false, then $\neg$A is true, and if A is true, then $\neg$A is false).

9 Prove that x$\in \mathbb{N}$ is a cube $\Leftrightarrow \sqrt[3]{x} \in \mathbb{N}$.

10 Prove that w $\in$ L1 $\wedge$ w $\in$ L2 $\Leftrightarrow$ L1 $\cap$ L2 $\neq \emptyset$.

Finally, we may use universal and existential quantification. Universal quantification states that for all members of a set, some predicate holds. For instance, we may define a set as follows:

$$\texttt{M4} = \{\texttt{n} \mid \texttt{n} \in \mathbb{N} \wedge \texttt{n is a multiple of 4}\}$$

We can make statements about all the members of `M4` such as:

$$\forall\ \texttt{n} \in \texttt{M4}\ \texttt{n = 2k, where k} \in \mathbb{N}$$

This states that all elements in `M4` are even. To prove a statement using universal quantification, we must argue that the statement is true for an arbitrary member of the set. Let us prove the above statement:

`Let x be an arbitrary member of M4.`

$$\begin{aligned} \texttt{x} \in \texttt{M4} &\Rightarrow \texttt{x = 4h} \\ &\Rightarrow \texttt{x = 2*2*h} \\ &\Rightarrow \texttt{x = 2k, where k = 2h} \end{aligned}$$

The chain of implications establish that `x` is an even number. Given that `x` is an arbitrary element of `M4`, we may conclude that all elements of `M4` are even.

Existential quantification states that there exists a member of the set for which a predicate holds. For example, we may make the following statement:

$\exists$ `n` $\in$ `M4 n is a multiple of 5`

This statement says that there exists an element of `M4` that is a multiple of 5. To prove a statement using existential quantification, we must demonstrate that a specific member of the set satisfies the predicate. Let us prove the statement above:

```
Let x = 20.

x = 20 ⇒ x = 4 * 5
       ⇒ x ∈ M4 ∧ x is a multiple of 5
```

The implications establish that 20 is a member of `M4` and that it is a multiple of 5. This means that the statement above holds. There is at least one value `n` for which the predicate holds.

13 Mathematical Induction Proofs

Formally, we define the set of natural numbers, $\mathbb{N}$, as follows:

```
A natural number is either:
  1. 0
  2. n + 1, where n ∈ ℕ
```

In essence, the definition states that 0 is a natural number and that $n \in \mathbb{N} \Rightarrow n+1 \in \mathbb{N}$. We say that 0 is a base instance of $\mathbb{N}$. That is, it is a value known to be in the set. A set, of course, may have more than one base value. Values that are not base values (let us call them inductive values) must be constructed from other elements in the set. The following chain of implications establishes that 5 is a natural number:

```
0 ∈ ℕ ⇒ 1 ∈ ℕ
      ⇒ 2 ∈ ℕ
      ⇒ 3 ∈ ℕ
      ⇒ 4 ∈ ℕ
      ⇒ 5 ∈ ℕ
```

Consider proving that a predicate (or statement) is true for all the natural numbers. What is required? The definition of a natural number defines two subtypes: the base instance and inductive instances. This suggests that such a proof must have two parts. The first part establishes that the predicate

holds for the base value (i.e., 0). This is called the base case (of the proof). The second part must establish that the predicate holds for the inductive instances. This is called the inductive step. The second part, however, requires special care. We cannot exhaustively prove the predicate for all inductive instances:

```
Prove P(1) holds
Prove P(2) holds
Prove P(3) holds
Prove P(4) holds
     ⋮
```

Clearly, we would never finish a proof because the natural numbers are infinite. What can we do? We must rely on the implication suggested by the data definition:

$$\texttt{n} \in \mathbb{N} \Rightarrow \texttt{n + 1} \in \mathbb{N}$$

What does this suggest? It suggests that we must show the following implication holds:

$$\texttt{P(n) holds} \Rightarrow \texttt{P(n+1) holds}$$

The good news is that we know how to prove an implication. In this case, we assume `P(n)` holds, and we must show that `P(n+1)` holds. `P(n)` is called the inductive hypothesis. This is the principle of *mathematical induction*. We may summarize it as follows:

1. Prove the base case.
2. The inductive step:
   a. State, `P(k)`, the inductive hypothesis.
   b. State, `P(k+1)`, what must be proven.
   c. Assume `P(k)` is true and prove `P(k+1)`.

It is important to note that the inductive hypothesis is not valid just for `k`. It is valid for all values in `[0..k]`.

13.1 Computing $\texttt{n}^2$

The `FSM` program displayed in Fig. 8 is a function that takes as input a natural number `n` and that returns $\texttt{n}^2$. Its body is a conditional expression that tests if `n` is 0. If so, it returns 0. Otherwise, it returns the sum of the next odd number, `2n - 1`, and the result of recursively processing `n - 1`. If you run the program, the unit tests pass. Do you believe it works for all natural numbers?

Fig. 8 A function to compute $\mathtt{n}^2$

```
#lang fsm

;; natnum → natnum
;; Purpose: Compute the square of the given natnum
(define (square n)
  (if (= n 0)
      0
      (+ (sub1 (* 2 n)) (square (sub1 n)))))

;; Tests
(check-equal? (square 0)   0)
(check-equal? (square 5)   25)
(check-equal? (square 100) 10000)
```

We can develop a proof by induction to establish that the function `square` works for all natural numbers (assuming we have enough memory to carry out the computation). We start by stating what we wish to prove:

Theorem 1 *(square n) returns* n^2.

The next step is to establish the base case:

Proof
Base Case: n = 0
If n = 0, then `(square n)` = `(square 0)` returns $0 = 0^2 = \mathrm{n}^2$. □

Having established the basis for our proof, we develop the inductive step:

Proof
Inductive Step:
Assume: `(square k)` returns k^2, for $\mathrm{n} = \mathrm{k} \geq 0$.

Show that: `(square (add1 k))` returns `(add1 k)`^{2}.

$\mathrm{k} \geq 0 \Rightarrow$ `(add1 k)` > 0
$\Rightarrow$ `(square k+1)` returns `(+ (sub1 (* 2 (add1 k))) (square k))`
$\Rightarrow$ `(square k+1)` returns `(+ (sub1 (+ (* 2 k) 2))` $\mathtt{k}^2$`)`
$\Rightarrow$ `(square k+1)` returns `(+ (+ (* 2 k) 1))` $\mathtt{k}^2$`)`
$\Rightarrow$ `(square k+1)` returns `(add1 k)`^{2} □

The inductive hypothesis assumes that the function works for a valid input greater than or equal to 0. We must prove that the function works for `k + 1`. The chain of implications start by observing that `k + 1` is greater than 0. This means the function returns the value of the else-branch of its conditional. This value is $\mathtt{k}^2$ `+ 2k + 1` $= (\mathrm{k} + 1)^2$. Thus, we may conclude that the function works for all natural numbers.

Fig. 9 A function to compute `n!`

```
#lang fsm

;; natnum → natnum
;; Purpose: Compute n!
(define (fact n)
  ;; natnum natnum → natnum
  ;; Purpose: Compute the factorial of the given natural number
  ;; Accumulator Invariant
  ;;    accum = Π_{i=j+1}^{n} i
  (define (helper j accum)
    (if (= j 0)
        accum
        (helper (sub1 j) (* j accum))))
  (helper n 1))

(check-equal? (fact 0)  1)
(check-equal? (fact 5)  120)
(check-equal? (fact 10) 3628800)
```

13.2 Computing `n!`

The `FSM` program displayed in Fig. 9 computes the factorial of a natural number, `n`, using accumulative recursion. The function `fact` takes as input a natural number, and its body calls the local auxiliary function `helper`. The function `helper` takes as input two natural numbers and returns a natural number. It computes the factorial of the first given number using an accumulator (the second given number). These inputs, respectively, start at `n` and 1. If the first given number, `j`, is 0, the accumulator, `accum`, is returned. Otherwise, `j - 1` is recursively processed with the product of `j` and `accum` as the new value of the accumulator. If you run the program, all the unit tests pass. This gives us cautious optimism that the program works. However, do you believe it works for all natural numbers?

Proving that this program works for all natural numbers hinges on proving that the accumulator invariant, `accum` $= \Pi_{i=j+1}^{n} i$, holds every time `helper` is called. Note that:

$$\Pi_{i=1}^{n}\ i \texttt{ = 1 * 2 * 3 * ... * n-1 * n}$$

How do we know that proving the accumulator invariant holds for every call to `helper` suffices to establish program correctness? Consider when `helper` halts. This occurs when `j = 0`. If the invariant holds, we have:

$$\begin{aligned} & \texttt{j = 0} \wedge \texttt{accum} = \Pi_{i=j+1}^{n}\ i \\ \Rightarrow\ & \texttt{accum} = \Pi_{i=0+1}^{n}\ i \\ \Rightarrow\ & \texttt{accum} = \Pi_{i=1}^{n}\ i \\ \Rightarrow\ & \texttt{accum = n!} \end{aligned}$$

Thus, if the invariant holds every time `helper` is called, `n!` is returned.

Fig. 10 Proof that the accumulator invariant holds when `helper` is called

Theorem 2 *accum* $= \Pi_{i=j+1}^{n} i$ *holds very time* `helper` *is called*

Proof

Base Case: n = 1

If `helper` is only called once then according to the code it must be the case that `j = 0` and `accum = 1`. This means that:

$$\begin{aligned} \Pi_{i=j+1}^{n} i &= \Pi_{i=1}^{0} i \\ &= 1 \\ &= \text{accum} \end{aligned}$$

Therefore, the invariant holds for the base case.

Inductive Step

Assume: accum $= \Pi_{i=j+1}^{n} i$ holds for n = k ≥ 1 calls to `helper`

We must show: accum $= \Pi_{i=j}^{n} i$ holds for k+1 calls to `helper`

The only way a `(k + 1)`$^{\texttt{th}}$ call is made is if `k` $\neq 0$.

This means that the condition in the code is false and the value returned is:

`(helper (sub1 j) (* j accum))`

= `(helper (sub1 j) (*` `j` $\Pi_{i=j+1}^{n} i$`))`, by inductive hypothesis

= `(helper (sub1 j)` $\Pi_{i=j}^{n} i$`)`

Note that we precisely know the values `helper` is given for the `k + 1` call. Observe that in the `k + 1` call the parameter `j` takes the value of `j - 1`. `j - 1`, however, has not been processed. This means that in our last statement above every `j` must be substituted with `j + 1` to reflect the last value processed in relation to the new value of `j`. This yields:

(helper j $\Pi_{i=j+1}^{n} i$)

Therefore, the invariant holds for the `k + 1` call. □

To prove the invariant always holds, we develop a proof by induction on the number of times `helper` is called. The proof is displayed in Fig. 10. It is noteworthy that the base case is `n = 1` (not 0), because when `fact` is called, the minimum number of times `helper` is called is 1. To clarify the end of the inductive step, carefully consider the transition from the `kth` call to the `(k + 1)th` call to `helper`. For the `kth` call, the invariant is expressed in terms of the value of `j`. The value of `j` changes to `j-1` when control is transferred to the `(k + 1)th` call. The value that was called `j` in the `kth` call is `j+1` in the `(k+1)th` call. For example, assume that for the `kth` call, `j` is 5. The invariant is expressed in terms of the current value of `j` (i.e., 5). That is, the validity of the invariant depends on 5. For the `(k + 1)th` call, `j` becomes 4. The validity of the invariant, however, still depends on 5. That is, in the `(k + 1)th` call, the validity of the invariant depends on `j + 1` (i.e., 5).

11 Prove that the following `FSM` program computes the sum of the first `n` natural numbers:

```
#lang fsm

;; natnum → natnum
;; Purpose: Compute the sum of the first n natural
;;          numbers
(define (sum-natnums n)
  (if (= n 0)
      0
      (+ n (sum-natnums (sub1 n)))))

(check-equal? (sum-natnums 0) 0)
(check-equal? (sum-natnums 5) 15)
(check-equal? (sum-natnums 20) 210)
```

12 The Fibonacci numbers are defined as follows:

```
A Fibonacci number is either:
  1. 0
  2. 1
  3. the sum of the 2 previous Fibonacci numbers
```

Prove that the following `FSM` program computes the `nth` Fibonacci number:

```
#lang fsm

;; natnum → natnum
;; Purpose: Compute the nth Fibonacci number
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (sub1 n)) (fib (- n 2)))))

(check-equal? (fib 0)  0)
(check-equal? (fib 1)  1)
(check-equal? (fib 5)  5)
(check-equal? (fib 10) 55)
```

13 An interval is a set of numbers defined as follows:

```
An interval is two integers, low and high, such that
it is either:
  1. empty (i.e., low > high)
  2. [[low..high-1]..high]
```

The data definition states that an interval is empty when `low` > `high`. `[5..4]`, for example, is an empty interval. When an interval is not empty, it consists of the subinterval `[low..high-1]` and `high`. The nonempty interval `[12..20]` = the numbers in `[12..19]` ∪ {20}.

Prove that the following `FSM` program computes the sum of the integers in the given interval:

```
#lang fsm

;; natnum → natnum
;; Purpose: Compute the sum of the integers in the
;;          given interval
(define (sum-interval low high)
  ;; integer integer → integer
  ;; Purpose: Compute the sum of the integers in
  ;;          [low..k]
  ;; Accumulator Invariant
  ;;    acc = Σ_{i=k+1}^{high} i
  (define (helper k acc)
    (if (< k low)
        acc
        (helper (sub1 k) (+ k acc))))
  (helper high 0))

(check-equal? (sum-interval 5 4)  0)
(check-equal? (sum-interval 1 3)  6)
(check-equal? (sum-interval -4 2) -7)
```

14 Pigeonhole Principle Proofs

Imagine that in a colony of pigeons, you have 50 pigeons and 45 pigeonholes. Clearly, it is impossible to place each pigeon in its own pigeonhole. Some pigeons will have to share a pigeonhole – an uncomfortable fate for the unlucky pigeons. Observe that a one-to-one function from the set of pigeons to the set of pigeonholes does not exists. This observation leads to *the pigeonhole principle*.

Theorem 2 *`A` and `B` are finite sets* ∧ `|A|` > `|B|` ⇒ ∄ *a one-to-one function from `A` to `B`.*

The symbol ∄ denotes the existential quantifier *does not exist*.

We shall prove the theorem by induction on `n = |B|`.

Proof
Base Case: n = 0 (i.e., `B` = ∅)
There is no function from `A` to `B`, because nothing can be mapped to elements of `B`. If there is no function from `A` to `B`, then there is no one-to-one function from `A` to `B`.
Inductive Step
Assume: `f: A` → `B` is not a one-to-one function such that `|A|` > `|B|`, `|B|` ≤ `n`, and `n` ≥ 0.
We must show: `f: A` → `B` is not a one-to-one function such that `|A|` > `|B|` = `n+1`.

Observe that `A` has at least two elements because `n+1` > 0. Pick two distinct arbitrary elements, `a` and `b`, from `A`. If `f(a)` = `f(b)`, then `f` is not one-to-one (because two distinct elements of `A` map to the same element in `B`). If `f(a)` ≠ `f(b)`, suppose that `a` is the only element mapped to `f(a)`. Consider the sets `A' = A - {a}` and `B' = B - {f(a)}` and a function `f'` such that ∀ `x` ∈ `A' f'(x) = f(x)`. The inductive hypothesis applies because `|B'|` = `n` and `|A'|` > `|B'|`. This means that there are two distinct elements in `A'` that are mapped by `f'` to the same element of `B'`. Given that `f` agrees with `f'` on all elements, it follows that `f` is not one-to-one. □

The pigeonhole principle may be a rather simple fact, but it is incredibly versatile. It has been used in a great number of proofs. We present a straightforward application of the pigeonhole principle.

Theorem 3 *Let `G` be a graph with `n` nodes. Any path with `n` edges has a repeated node.*

Proof Every edge connects two nodes (not necessarily distinct nodes). This means that `n` edges connect `n+1` nodes. By the pigeonhole principle, there is no one-to-one function from the nodes in the path (the pigeons) to the nodes in the graph (the pigeonholes). Therefore, we may conclude that there is at least one node repeated in the path. □

14 Prove that in a room with 367 people, there is at least one pair of persons with the same birthday.

15 Prove that in a room with `n` > `1` people that may shake hands with each other, there is always a pair of persons that shake hands with the same number of people.

16 Prove that for every 27-word sequence in this textbook, at least 2 words start with the same letter.

15 Proofs by Contradiction

A proof by contradiction proves that a statement holds by showing that assuming that the statement is false is absurd. That is, it leads to a contradiction. It is also known as reductio ad absurdum. It is based on the observation that a statement cannot be both true and false. That is, a statement, `S`, and its contradiction, $\neg$`S`, cannot both be true. A proof by contradiction establishes that the negation of a statement leads to such a contradiction. That is, assuming `S` is false leads to concluding that a statement `A`, which we know to be false, is true. This means that our assumption must be wrong and, therefore, `S` must be true. As an illustrative example, we prove that $\sqrt{2}$ is an irrational number.

Theorem 4 $\sqrt{2}$ *is an irrational number.*

Proof
Assume $\sqrt{2}$ is a rational number.

If $\sqrt{2}$ is rational, then it can be expressed as a fraction, $\frac{a}{b}$, in lowest terms for `a,b` $\in \mathbb{Z}$. Observe that at least one of `a` and `b` must be odd. Consider:

$$\frac{a}{b} = \sqrt{2}$$
$$\frac{a^2}{b^2} = 2$$
$$a^2 = 2b^2$$

This means that a^2 is even. Observe that this also means that `a` is even because `a * a` must be divisible by 2 (if `a` were odd, then `a * a` would not be divisible by 2).

Given that `a` is even and $\frac{a}{b}$ is in lowest terms, `b` must be odd (otherwise, `a` and `b` have 2 as a common factor).

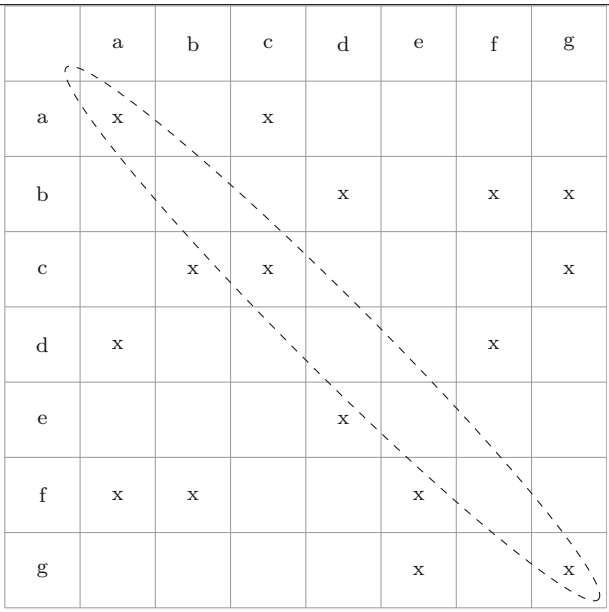
Observe that a^2 is a multiple of 4:

$$\begin{aligned} a^2 &= a * a \\ &= 2j * 2j \\ &= 4j^2 \end{aligned}$$

This means that $2b^2$ is a multiple of 4. We observe that `b` must be even given that if it were odd, `b` could equal 3 which means $2b^2$ is not a multiple of 4 (i.e., $2(3)^2 = 18$ which is not a multiple of 4).

It is impossible, however, for `b` to be both odd and even. Therefore, our assumption cannot be true, and we may conclude that $\sqrt{2}$ is an irrational number. □

Fig. 11 The diagonal set of a relation



	a	b	c	d	e	f	g
a	x		x				
b				x		x	x
c		x	x				x
d	x					x	
e				x			
f	x	x			x		
g					x		x

17 Prove by contradiction that the sum of two even numbers is always even.

18 Prove by contradiction that there are no values `x` and `y` such that `24y + 12z = 1`. *Hint*: If you assume that two such values exist, then you may use algebra.

16 Diagonalization Proofs

A binary relation on a set `A` may be visualized as a matrix whose rows and columns are labeled with the elements of `A`. If `i` is related to `j`, then the entry in row `i` and column `j` contains an `x`. Figure 11 is the visualization of a binary relation on `A` = {`a b c d e f g`}. We can observe, for example, that `a` is related to `a` and `c`, that `e` is only related to `d`, and that `f` is not related to `c`, `d`, `f`, and `g`.

The main diagonal of the binary relation visualization defines two sets: those elements that are related to themselves and those elements that are

not related to themselves. We may formally define these sets for a relation R as follows:

$$D = \{a \mid (a, a) \in R\}$$

$$\hat{D} = \{a \mid (a, a) \notin R\}$$

We refer to D as the diagonal set and $\hat{D}$ as the complement of the diagonal set. In Fig. 11, the main diagonal is highlighted with a dashed oval. We have that the diagonal and the complement of the diagonal sets are:

$$D = \{a\ c\ g\}$$

$$\hat{D} = \{b\ d\ e\ f\}$$

The *diagonalization principle* states that $\hat{D}$ is not equal to any row in the visualization. In other words, if R_a is the set of elements a is related to, then $\hat{D} \neq R_a$. You can visually confirm that $\hat{D}$ for the relation in Fig. 11 is different from every row (i.e., different from R_a–R_g).

The diagonalization principle is used as part of a proof by contradiction. A statement is assumed to be true, and diagonalization is used to develop a contradiction. To illustrate diagonalization in practice, we shall prove that the real numbers in (0..1) are uncountable. We shall use a well-known fact for computer scientists: every real number in (0..1) may be written as a binary number of infinite length.

Theorem 5 *The set of real numbers in (0..1) is uncountable.*

Proof
Assume the set of real numbers in (0..1) is countable.

This means there exists a program to print these numbers that eventually prints any arbitrary binary digit of any real number in (0..1). The printing of these real numbers looks as follows:

```
1. .0 0 0 1 0 1 1 ...
2. .1 1 0 1 0 0 0 ...
3. .0 1 0 0 0 0 0 ...
4. .0 1 1 0 0 0 1 ...
5. .0 0 0 0 1 1 0 ...
6. .1 1 1 1 1 1 1 ...
7. .1 1 0 0 0 0 0 ...
     ⋮
```

Observe that the binary digits form a matrix. Consider the real number represented by $\hat{D}$. $\hat{D}$ cannot ever equal an arbitrary row i in the matrix of printed numbers because their ith bits differ. This means that the real number represented by $\hat{D}$ is never printed. This contradicts the assumption made that the set of real numbers in (0..1) is countable and, therefore, the set of real numbers in (0..1) is uncountable. □

Recall that in the previous chapter, we tabled the discussion on whether or not $\mathbb{R}$ and $\mathbb{Z}$ are equinumerous. Observe that the proof above means that there is no bijection between $\mathbb{N}$ and $\mathbb{R}$ (also see problem 20). That is, $\mathbb{N}$ and $\mathbb{R}$ are not equinumerous. We know from the previous chapter that $\mathbb{N}$ and $\mathbb{Z}$ are equinumerous. Therefore, there does not exist a bijection between $\mathbb{R}$ and $\mathbb{Z}$ meaning that these sets are not equinumerous.

19 Prove that $2^{\mathbb{N}}$ is uncountable.

20 Prove that $\mathbb{R}$ is uncountable.

21 Prove that if $|\mathrm{A}| = \infty$, then 2^{A} is uncountable.

Part II
Regular Languages

Chapter 4
Regular Expressions

In everyday life, we think of English as a language that consists of a set of words. Each word is a string of juxtaposed letters found in the Roman alphabet: `[a..z]`. As a computer science student and reader of this textbook, you are familiar with other languages. For example, you are familiar with various programming languages, and you are familiar with the binary language. The words of the binary language, for example, are binary numbers. Each binary number is a finite number of juxtaposed elements found in the binary alphabet: `{0 1}`.

Defining English words as strings is, of course, a representation choice. Recall, for example, that in `FSM` a word is represented as a list of alphabet elements. Consider three different representations of the word *cat*:

```
English: "cat"
    FSM: '(c a t)
 Binary: 011000110110000101110100
```

In binary, the word *cat* is obtained by juxtaposing the `ascii` code for each of its letters. The representations of *cat* are different, but the same concept is being represented. That is, the representation does not change the meaning. A language, therefore, may be represented as a set of strings, a set of lists, or a set of binary numbers. Regardless of the representation, the same words in the English language are represented.

Languages are represented using a finite representation. If a language is finite, listing all the words in the language is a finite representation. Many interesting languages, however, are not finite. That is, they contain an infinite number of words. A finite representation for infinite languages is needed because all the words in an infinite language cannot be listed. A finite representation for a language must be written using a finite number of symbols and must be different from the representation used for any other language. If

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_4

Σ is an alphabet used to write the finite representations of languages, then all possible finite language representations are defined as Σ^*. This means that the language of finite language representations is countably infinite (they can be printed in alphabetical order like the words in a complete English dictionary). 2^{Σ^*}, however, is uncountable. So, there are a countable number of finite language representations and an uncountable number of languages to represent. Therefore, a finite representation for each language does not exist. The best that we can achieve is to develop a finite representation of some interesting languages. As long as a representation is finite, the majority of languages cannot be represented.

17 Defining Languages Using Union and Concatenation

17.1 Constructors

We start by considering languages formed by the union or the concatenation of words in two (not necessarily distinct) languages. Such languages may be finitely represented using regular expressions. A regular expression, over an alphabet Σ, is an `FSM` type instance:

```
1. (empty-regexp)
2. (singleton-regexp "a"), where a∈Σ
3. (union-regexp r1 r2),  where r1 and r2 are regular
                          expressions
4. (concat-regexp r1 r2), where r1 and r2 are regular
                          expressions
5. (kleenestar-regexp r), where r is a regular expression
```

Each regular expression subtype is built using a distinct constructor. The language of a regular expression, `r`, is denoted by `L(r)`. It contains all the words that can be generated with `r`. A language that is described by a regular expression is called a *regular language*.

The first regular expression describes the following language:

```
L((empty-regexp)) = {EMP}
```

That is, it is a language that only contains the empty word.

The constructor `singleton-regexp` is used to build a regular expression for any element in Σ. It takes as input a string representing an element in Σ. A singleton regular expression describes the following language:

```
L((singleton-regexp "a")) = {a}
```

That is, it is the language that only contains `a`.

If `r1` and `r2` are regular expressions, then a regular expression for `L(r1)` $\cup$ `L(r2)` is built using `union-regexp`. It describes the following language:

```
L((union-regexp r1 r2)) = {w | w∈L(r1) ∨ w∈L(r2)}
```

That is, it represents the language that contains all the words in `L(r1)` and all the words in `L(r2)`.

If `r1` and `r2` are regular expressions, then `concat-regexp` builds a regular expression for `L(r1)L(r2)`. It describes the following language:

$$\texttt{L((concat-regexp r1 r2))} = \{\texttt{w}_1\texttt{w}_2 \mid \texttt{w}_1 \in \texttt{L(r1)} \wedge \texttt{w}_2 \in \texttt{L(r2)}\}$$

That is, it is the language that contains all words constructed by concatenating a word in `L(r1)` and a word in `L(r2)`.

If `r` is a regular expression, then a regular expression for zero or more concatenations of words in `L(r)` is built using `kleenestar-regexp`. It represents the following language:

$$\texttt{L((kleenestar-regexp r))} = \{\{\texttt{EMP}\} \cup \{\texttt{w}_1\texttt{w}_2 \ldots \texttt{w}_n \mid \texttt{w}_1, \texttt{w}_2, \ldots, \texttt{w}_n \in \texttt{L(r)} \wedge \texttt{n} \geq 1\}\}$$

That is, it is the language that contains all words constructed by concatenating zero or more words in `L(r)`. Nothing else is a regular expression.[6]

17.2 Error Messages

`FSM` provides informative error messages to help you overcome the misuse of constructors. When a constructor is misused, an error is thrown. This is a sampling of `FSM` error messages for misuse of regular expression constructors:

```
> (union-regexp 2 (singleton-regexp 'w))
the input to the regexp #(struct:singleton-regexp w)
must be a string
> (union-regexp (empty-regexp) 3)
3 must be a regexp to be a valid second input to
union-regexp #(struct:empty-regexp) 3
> (concat-regexp 3 (empty-regexp))
3 must be a regexp to be a valid first input to
concat-regexp 3 #(struct:empty-regexp)
> (kleenestar-regexp "A U B")
"A U B" must be a regexp to be a valid input to
kleenestar-regexp
> (singleton-regexp 1)
the input to the regexp #(struct:singleton-regexp 1)
must be a string
```

[6] In a future chapter, we shall discover the need for one more regular expression variety: one to describe the empty language.

17.3 Regular Expression Selectors and Predicates

The constructors for each regular expression subtype have been discussed. To process a regular expression, however, we need to know the selector functions to access its components, and we need predicates to distinguish between regular expression subtypes. The `FSM` selector functions for regular expressions are:

singleton-regexp-a: Extracts the embedded string
kleenestar-regexp-r1: Extracts the embedded regular expression
union-regexp-r1: Extracts the first embedded regular expression
union-regexp-r2: Extracts the second embedded regular expression
concat-regexp-r1: Extracts the first embedded regular expression
concat-regexp-r2: Extracts the second embedded regular expression

The following predicates are defined to distinguish among the regular expression subtypes:

```
empty-regexp?     singleton-regexp?     kleenestar-regexp?
union-regexp?     concat-regexp?
```

Each selector and predicate consumes a regular expression. The predicates return a Boolean. They return true if the input is a regular expression of the subtype tested. Otherwise, they return false. The selectors return a regular expression used to build an instance of a regular expression except `singleton-regexp-a` that returns the string used to build the regular expression instance.

Armed with the constructors, selectors, and predicates for regular expressions, we can write a template for functions on a regular expression:

```
;; regexp ... → ...
;; Purpose: ...
(define (f-on-regexp rexp ...)
  (cond [(empty-regexp? rexp) ...]
        [(singleton-regexp? rexp)
         ...(f-on-string (singleton-regexp-a rexp))...]
        [(kleenestar-regexp? rexp)
         ...(f-on-regexp (kleenestar-regexp-r1 rexp))...]
        [(union-regexp? rexp)
         ...(f-on-regexp (union-regexp-r1 rexp))...
         ...(f-on-regexp (union-regexp-r2 rexp))...]
        [else ...(f-on-regexp (concat-regexp-r1 rexp))...
              ...(f-on-regexp (concat-regexp-r2 rexp))...]))
```

The function template reflects the structure of regular expressions. That is, it puts forth the use of structural recursion to process a regular expression. It suggests processing the embedded string for a singleton regular expression. For a Kleene star regular expression, it suggests recursively processing

the embedded regular expression. Finally, for union and concatenation regular expressions, it suggests recursively processing both embedded regular expressions.

17.4 Observers

A regular expression describes how a word in its language is generated. That is, it describes a generating algorithm. Unlike most algorithms you have studied, a regular expression may describe a *nondeterministic* algorithm. That is, it may describe an algorithm whose result is not fully predictable. `FSM`, however, does not burden programmers with implementing nondeterminism. It provides functions that allow you to generate words from a given type of regular expression. These are briefly outlined as follows:

(pick-regexp r): Nondeterministically returns a nested sub-`regexp` from the given `union-regexp`. This includes any nested `union-regexp`s in a chain of `union-regexp`s. For example, if the `union-regexp` is:

```
(union-regexp r1 (union-regexp r2 (union-regexp r3 r4)))
```

the selected `regexp` may be any of `r1`–`r4`.

(pick-reps n): Nondeterministically generates a natural number in [0..n].

Consult the `FSM` documentation in `DrRacket` for more detailed descriptions. For convenience, `FSM` also provides the following function to generate a word from a `singleton-regexp`:

`(convert-singleton r)`: Converts the given `singleton-regexp` to a word of length 1 containing r's nested symbol or number.

Finally, `FSM` provides an observer for a printable version of a regular expression. By printable, interpret a string fit for humans to read. The following table outlines the printable forms of regular expressions:

Regular expression	Printable form
`(empty-regexp)`	the value of `EMP`
`(singleton-regexp a)`	`" a"`
`(union-regexp r1 r2)`	(string-append (printable-regexp r1) "U" (printable-regexp r2))
`(concat-regexp r1 r2)`	(string-append (printable-regexp r1) (printable-regexp r2))
`(kleenestar-regexp r)`	(string-append (printable-regexp r1) `"*"`)

The `FSM` function `printable-regexp` returns a string representing the regular expression it is given as input. The following interactions illustrate how `printable-regexp` works:

```
> (printable-regexp (empty-regexp))
"ϵ"
> (printable-regexp (singleton-regexp "z"))
"z"
> (printable-regexp (union-regexp
                     (singleton-regexp "z")
                       (union-regexp
                         (singleton-regexp "1")
                         (singleton-regexp "q"))))
"(z U (1 U q))"
> (printable-regexp (concat-regexp (singleton-regexp "i")
                                   (singleton-regexp "i")))
"ii"
> (printable-regexp (kleenestar-regexp
                     (concat-regexp
                       (singleton-regexp "a")
                       (singleton-regexp "b"))))

"(ab)*"
```

18 Programming with Regular Expressions

Regular expressions are values in `FSM`. As such, you may design their computation using a top-down or a bottom-up divide-and-conquer approach. The idea is to define a regular expression by parts. Just like a program is composed of one or more functions or methods, a regular expression is composed of one or more regular expressions. In this section, we explore how to design and write regular expressions.

18.1 All Words Ending with an `a`

We can now build a finite representation for some infinite languages. For example, consider the following language over Σ = {a b}:

```
L = {w | w ends with an a}
```

Can we program a regular expression for `L`? We shall follow a top-down design. Every word in `L` must have at least one `a` at the end. Before the last `a`, there can be an arbitrary number of `a`s and `b`s. Assuming a regular expression can be developed for both parts, the regular expression for `L` that concatenates the languages for each part may be written as follows:

```
(define ENDS-WITH-A (concat-regexp AUB* A))
```

Fig. 12 The FSM program for L = {w | w ends with an a}

```
#lang fsm

(define A (singleton-regexp "a"))

(define B (singleton-regexp "b"))

(define AUB (union-regexp A B))

(define AUB* (kleenestar-regexp AUB))

(define ENDS-WITH-A  (concat-regexp AUB* A))

(check-equal? (printable-regexp ENDS-WITH-A) "(a U b)*a")
```

The regular expression A must represent a. This is defined as follows:

```
(define A (singleton-regexp "a"))
```

AUB* must represent an arbitrary number of elements. This suggests defining a kleenestar-regexp. This may be done as follows:

```
(define AUB* (kleenestar-regexp AUB))
```

AUB represents a choice between a word in L(A) and a word in L(B). Having a choice suggests defining a union-regexp:

```
(define AUB (union-regexp A B))
```

Finally, B represents the singleton regular expression for "b". This is done as follows:

```
(define B (singleton-regexp "b"))
```

The complete program for a regular expression for L is displayed in Fig. 12.

18.2 Binary Numbers

Consider the following language:

BIN-NUMS = {w | w is a binary number without leading zeroes}

Although the above definition may sound clear, it is lacking. It does not provide any details about the structure of the binary numbers in the language nor any indication on how to build such numbers. We shall attempt to formally define BIN-NUMS using a regular expression using a bottom-up divide-and-conquer approach.

Fig. 13 A program defining binary numbers without leading zeroes

```
#lang fsm

(define ZERO (singleton-regexp "0"))

(define ONE  (singleton-regexp "1"))

(define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))

(define STARTS1 (concat-regexp ONE 0U1*))

(define BIN-NUMS (union-regexp ZERO STARTS1))

(check-equal? (printable-regexp BIN-NUMS) "(0 U 1(0 U 1)*)")
```

18.2.1 Implementing a Regular Expression

Based on the problem statement, the following observations are made:

1. Σ = {0 1}.
2. The minimum length of a binary number is 1.
3. A binary number with a length greater than 1 cannot start with 0.

The second observation informs us that the empty regular expression is not part of the language. The third observation informs us that there are two subtypes of binary numbers in the set: 0 and those starting with 1 followed by an arbitrary number of 0s and 1s.

The simplest regular expressions needed are for the elements of Σ. These are all singleton regular expressions:

```
(define ZERO (singleton-regexp "0"))

(define ONE  (singleton-regexp "1"))
```

An arbitrary number of 0s and 1s may be represented using a union and a Kleene star regular expression:

```
(define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))
```

With the above definition, a regular expression for 1 followed by an arbitrary number of 0s and 1s is:

```
(define STARTS1 (concat-regexp ONE 0U1*))
```

Finally, BIN-NUMS may be implemented by a union regular expression to provide a choice among the subtypes:

```
(define BIN-NUMS (union-regexp ZERO STARTS1))
```

Figure 13 displays the complete program to define BIN-NUMS. When you run the program, the test passes.

1 Implement a regular expression for the language that contains all words that start with one or more `a`s and end with a `b`.

2 Let Σ = {`a b`}. Implement a regular expression for the language that contains all words that have three `b`s.

3 Let Σ = {`a b`}. Implement a regular expression for the language that contains all words of even length.

4 Let Σ = {`0 1`}. Implement a regular expression for the language that contains all words that have a single `1`.

5 Implement a regular expression for the language that contains all words that represent the nonnegative integers.

6 Let Σ = {`a b`}. Implement a regular expression for the language that contains all words that have a length less than or equal to 3.

18.2.2 Generating `BIN-NUMS` Words

Compare the two formulations for `BIN-NUMS`:

```
BIN-NUMS = {w | w is a binary number without leading zeroes}

BIN-NUMS = (0 ∪ 1(0 ∪ 1)*)
```

Which do you believe is more useful? The truth is that both are useful. The first provides a quick understanding of what the language `BIN-NUMS` represents. It lacks, however, any description for constructing words in `BIN-NUMS`. In this regard, the second formulation is more useful. It describes an algorithm for constructing binary numbers without leading zeroes. Either generate 0 or generate 1 followed by an arbitrary number of `0`s and `1`s.

This means that a function to generate a random word in `BIN-NUMS` can and ought to be implemented. We shall follow the steps of the design recipe to write this program. To simplify the discussion, the maximum word length shall have a default value of 10. We allow, however, for the user to define the default maximum length through an optional parameter. Optional arguments are accumulated in a list whose name appears after a `.` in the function header. Based on this design idea, the next steps of the design recipe are outlined as follows:

```
;;  [natnum>0] → word
;; Purpose: Generate a binary number without leading
;;          zeroes
(define (generate-bn . n)
  (define MAX-KS-REPS (if (null? n) 10 (first n)))
        ⋮
 ...)
```

Observe that the list of optional arguments is called `n`. The signature specifies that an optional (in square brackets) natural number greater than 0 may be given as input. The function returns a word that according to the purpose statement is a binary number without unnecessary leading zeroes.

We are unable to write tests using `check-equals?` because words are generated from regular expressions that require a nondeterministic choice to be made. Specifically, a `union-regexp` requires a sub-`regexp` to be chosen, and a `kleene-star-regexp` requires the number of repetitions to be chosen. To validate such a function, we use property-based testing. That is, we shall test that the generated words have the expected properties. To write tests, we use `rackunit`'s `check-pred`. `check-pred` requires a predicate to test and input for the predicate. If the predicate holds, the test passes. If the predicate does not hold, then the test fails, and a failed test report is generated. Any word, `w`, generated by `generate-bn` must have the following properties:

1. `w` is a list (i.e., it cannot be `EMP`).
2. $1 \leq$ `(length w)`.
3. `w` is `'(0)` or `(first w)` is 1.
4. `w` only contains `0`s and `1`s.

To test that a generated word represents a binary number, we ignore the length limit. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Test if the given word is in L(BIN-NUMS)
(define (is-bin-nums? w)
  (and (list? w)
       (<= 1 (length w))
       (or (equal? w '(0)) (= (first w) 1))
       (andmap (λ (bit) (or (= bit 0) (= bit 1))) w)))

(check-equal? (is-bin-nums? '()) #f)
(check-equal? (is-bin-nums? '(0 0 0 1 1 0 1 0)) #f)
(check-equal? (is-bin-nums? '(0)) #t)
(check-equal? (is-bin-nums? '(1 0 0 1 0 1 1)) #t)
(check-equal? (is-bin-nums? '(1 1 1 0 1 0 0 0 1 1 0 1)) #t)
```

This predicate is used to write the tests for `generate-bn`. Anything returned by `generate-bn` must satisfy `is-bin-nums?`. The tests are:

```
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn 10))
(check-pred is-bin-nums? (generate-bn 25))
(check-pred is-bin-nums? (generate-bn 5))
```

Although the tests all look the same, they are not the same test. Recall that `(generate-bn)` is nondeterministic (i.e., the output value cannot be predicted). Therefore, each test above is for a different word (not necessarily distinct) returned by `generate-bn`.

`BIN-NUMS` is used in `generate-bn`'s body to generate valid binary numbers. The task of generating these is delegated to an auxiliary function as follows:

```
(gen-word BIN-NUMS)
```

As you may have already realized, `gen-word` must be able to process any `regexp` subtype that is part of the `BIN-NUMS`: `singleton-regexp`, `concat-regexp`, `union-regexp`, and `kleenestar-regexp`. The template for a function on a `regexp` is specialized. To process a `singleton-regexp`, `convert-singleton` is used. To process a `concat-regexp`, a word is generated using each embedded `regexp`, and they are appended. To process a `union-regexp`, a recursive call is made with one of the sub-`regexps` nondeterministically chosen by `pick-regexp`. To process a `kleenestar-regexp`, the number of binary numbers to generate is nondeterministically chosen using `pick-reps`. A list containing that number of binary numbers is generated and flattened. The result of this design, including locally defining `gen-word`, is displayed in Fig. 14.

7 Based on a regular expression, design and write a function to generate a word that starts with one or more `a`s and ends with a `b`.

8 Based on a regular expression, design and write a function to generate a word that represents a natural number.

19 Generating Words in the Language Defined by a Regular Expression

The development of `generate-bn` confirms that a regular expression, `r`, describes a construction algorithm for words in `L(r)`. This suggests that there is an algorithm to generate an arbitrary word in the language of an arbitrary

Fig. 14 A program to generate words in BIN-NUMS

```
#lang fsm

;; [natnum>0] → word
;; Purpose: Generate a binary number without leading zeroes,
;;          unless its 0, of length <= MAX-LENGTH
(define (generate-bn . n)

  (define MAX-KS-REPS (if (null? n) 10 (first n)))

  ;; regexp → word
  ;; Purpose: Generate a word representing a valid binary number,
  ;;          such that the number of Kleene star repetitions is
  ;;          in [0..MAX-KS-REPS]
  (define (gen-word r)
    (cond [(singleton-regexp? r) (convert-singleton r)]
          [(concat-regexp? r)
           (let [(w1 (gen-word (concat-regexp-r1 r)))
                 (w2 (gen-word (concat-regexp-r2 r)))]
             (append w1 w2))]
          [(union-regexp? r) (gen-word (pick-regexp r))]
          [(kleenestar-regexp? r)
           (flatten (build-list
                     (pick-reps MAX-KS-REPS)
                     (λ (i)
                       (gen-word (kleenestar-regexp-r1 r)))))]))
   (gen-word BIN-NUMS))

;; Tests
;; word → Boolean
;; Purpose: Test if the given word is a BIN-NUMS)
(define (is-bin-nums? w)
  (and (list? w)
       (<= 1 (length w))
       (or (equal? w '(0)) (= (first w) 1))
       (andmap (λ (bit) (or (= bit 0) (= bit 1))) w)))

(check-equal? (is-bin-nums? '()) #f)
(check-equal? (is-bin-nums? '(0 0 0 1 1 0 1 0)) #f)
(check-equal? (is-bin-nums? '(0)) #t)
(check-equal? (is-bin-nums? '(1 0 0 1 0 1 1)) #t)
(check-equal? (is-bin-nums? '(1 1 1 0 1 0 0 0 1 1 0 1)) #t)

(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn 10))
(check-pred is-bin-nums? (generate-bn 25))
(check-pred is-bin-nums? (generate-bn 5))
```

regular expression. As such, we ought to implement it. Given that regular expressions are native in `FSM`, it is a good choice for implementing such an algorithm.

19.1 Design Idea

The function takes as input a regular expression and an optional natural number for the maximum number of Kleene star repetitions. It returns a word. A constant, `MAX-KLEENESTAR-REPS`, is locally defined for the maximum number of Kleene star repetitions. If the optional natural number is not provided, the default value of the constant is arbitrarily defined to be 20. Building on our experience generating binary numbers, we shall also have a local function, say `generate`, to generate a word.

As suggested by the function template to process a `regexp`, `generate` must distinguish among the regular expression subtypes to generate the word. If the input is the empty regular expression, then the only word that may be generated is, `EMP`, the empty word. If given a singleton regular expression, then `convert-singleton` is used to generate a word of length 1 from the embedded string.

To process a Kleene star regular expression, a list of words is generated using the embedded `regexp`. The length of the list is nondeterministically chosen using `pick-reps`. Once generated, the list of words is filtered for empty words and flattened. If the resulting list is empty, then `EMP` is returned. Otherwise, the resulting list is returned.

To process a union regular expression, `pick-regexp` is used to nondeterministically select one of the expressions in the union. A recursive call is made with the selected regular expression to generate the word.

To process a `concat-regexp`, a word is generated using each of the embedded regular expressions. If both generated words are `EMP`, then `EMP` is returned. If either word is `EMP`, then the other word is returned. Otherwise, the two generated words are appended and returned.

19.2 Signature, Purpose, and Function Header

The signature, purpose statement, and function header collectively provide documentation that explains to any reader of the code what the function is expected to do. The next steps of the design recipe are satisfied as follows:

```
;; regexp [natnum] → word
;; Purpose: Generate a random word in the language
;;          of the given regexp such that the number
;;          of repetitions generated from a Kleene
;;          star regular expression does not exceed
;;          (max 20 reps).
(define (gen-word rexp . reps)
```

Observe that, once again, the optional argument is captured in a list (in this case, named `reps`).

19.3 Tests

To simplify the development of tests, we use `ENDS-WITH-A` from Sect. 18.1 and `BIN-NUMS` from Sect. 18.2.2. Given that we are designing a nondeterministic function, property-based testing is employed. This means we must design and implement a predicate for words ending with an `a` just like `is-bin-nums?` is designed for `BIN-NUMS`. If the given word is a list and has a length greater than or equal to 1 and its last element is an `a`, then it is a word in `L(ENDS-WITH-A)`. Following the steps of the design recipe yields this predicate:

```
;; word → Boolean
;; Purpose: Test if the given word is in ENDS-WITH-A
(define (is-ends-with-a? w)
  (and (list? w) (>= (length w) 1) (eq? (last w) 'a)))

(check-equal? (is-ends-with-a? '(a)) #t)
(check-equal? (is-ends-with-a? '(b b a)) #t)
(check-equal? (is-ends-with-a? '(a b b a b a)) #t)
(check-equal? (is-ends-with-a? '()) #f)
(check-equal? (is-ends-with-a? '(b b b)) #f)
(check-equal? (is-ends-with-a? '(a a a a b)) #f)
```

The tests for `gen-word` are:

```
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS 30))
(check-pred is-bin-nums? (gen-word BIN-NUMS 50))

(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A 18))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A 7))
```

Recall that each test is different because `gen-word` is a nondeterministic function that may produce a different result given the same input.

19.4 Function Body

The next step of the design requires writing the function's body. This is done by specializing the `cond`-expression in the template for functions on a regular expression. We independently present the design of each stanza.

For the empty regular expression, the only word that can be generated is `EMP`. The corresponding stanza is:

```
[(empty-regexp? rexp) EMP]
```

For a `singleton-regexp`, a word is generated using `convert-singleton`:

```
[(singleton-regexp? rexp) (convert-singleton rexp)]
```

For a `kleenestar-regexp`, the length of the word is nondeterministically chosen using `pick-reps`. A list of words of the chosen length, generated using the embedded regular expression, is filtered to remove all `EMP`s and flattened. If the flattened list is empty, then `EMP` is returned. Otherwise, the flattened list is returned. The required code is:

```
[(kleenestar-regexp? rexp)
 (let*
  [(reps (pick-reps MAX-KLEENESTAR-REPS))
   (low (flatten
         (filter
          (λ (w) (not (eq? w EMP)))
           (build-list
            reps
            (lambda (i)
             (gen-word (kleenestar-regexp-r1 rexp)))))))]
  (if (empty? low) EMP low))]
```

For a union regular expression, a word is generated by nondeterministically picking a regular expression from the options in the union and making a recursive call with the maximum number of repetitions:

```
[(union-regexp? rexp) (gen-word (pick-regexp rexp))]
```

For a concatenation regular expression, two words are generated using each embedded regular expression. The words are examined as described in the design idea to return the generated word. The default stanza of the conditional is:

```
[else
 (let [(w1 (gen-word (concat-regexp-r1 rexp)))
       (w2 (gen-word (concat-regexp-r2 rexp)))]
   (cond [(and (eq? w1 EMP) (eq? w2 EMP)) EMP]
         [(eq? w1 EMP) w2]
         [(eq? w2 EMP) w1]
         [else (append w1 w2)]))]
```

19.5 Running the Tests

Make sure that all the tests pass. If any tests fail, correct the errors by making sure you have followed all the steps of the design recipe as presented.

Take time to appreciate what has been achieved. A regular expression, simultaneously, is a description of, `L`, a regular language and a description of a construction algorithm for the words in `L`. That is, a construction algorithm specifies a language. Indeed, a construction algorithm is an elegant way of describing a language.

The function designed is so versatile that it is part of `FSM`. In `FSM`, it is called `gen-regexp-word`. This primitive nondeterministically generates a word in the language of the `regexp` that it is given as input. You may read its full description using `DrRacket`'s Help Desk (under the `Help` tab).

9 Let Σ = {`a b`}. Use `gen-regexp-word` to generate words with a number of `a`s that is divisible by 3.

20 Regular Expression Applications

Regular expressions capture a pattern for the construction of languages. As such, regular expressions are easily found in many areas of computer science and, indeed, in life. It is important to note that the term regular expression is used differently in different domains. That is, a regular expression is not always defined as in this chapter. Generally, they all have union, concatenation, and Kleene star operations. They, however, also include other operations. These other operations may provide the ability to describe languages that are not regular. The syntax, of course, also varies. Consider, for example, the following `Perl` code snippet:

```
$foo =~ m/fsm/
```

This expression evaluates to true if `$foo` contains `fsm`. Put differently, it evaluates to true if `$foo`'s value is a word in the following language:

$$\texttt{L} = \{\texttt{w} \mid x', y' \in \Sigma^* \wedge \texttt{w} = x'\texttt{fsm}y'\}$$

Clearly, `L` is a regular language. Regular expressions in `Perl`, however, are powerful enough to match languages that are not regular. Therefore, when speaking about regular expressions, it is important to be precise. In this textbook, we always mean regular expressions as defined in this chapter.

Regular expressions may be used to describe data such as internet addresses, proteins, decimal numbers, and patterns to search for in text among others. To illustrate the use of regular expressions, we explore the problem of generating passwords. As always, we follow the steps of the design recipe to write a password-generating function.

20.1 Data Definitions

A password is a string that:

- Has length ≥ 10
- Includes at least one lowercase letter
- Includes at least one uppercase letter
- Includes at least one special character: $, &, !, and *

Based on this definition, the sets for lowercase letters, uppercase letters, and special characters are defined as follows:

```
(define lowers '(a b c d e f g h i j k l m n o p q
                 r s t u v w x y z))
(define uppers '(A B C D E F G H I J K L M N O P Q
                 R S T U V W X Y Z))
(define spcls  '($ & ! *))
```

The corresponding sets of regular expressions are defined as:

```
(define lc (map
            (λ (lcl) (singleton-regexp (symbol->string lcl)))
            lowers))
(define uc (map
            (λ (ucl) (singleton-regexp (symbol->string ucl)))
            uppers))
(define spc (map
             (λ (sc) (singleton-regexp (symbol->string sc)))
             spcls))
```

Each set is traversed using `map`. The function given to `map` converts a symbol into a string and then creates a singleton regular expression.

How is a password defined? To create passwords, we need a regular expression. Once a word representing a valid password is generated, it can be

transformed into a string. The order in which lowercase letters, uppercase letters, and special characters appear is arbitrary. There must be, however, an, `L`, lowercase letter; a, `U`, uppercase letter; and a, `S`, special character. There are six different orderings these required elements may appear in:

```
L U S     U L S     S U L
L S U     U S L     S L U
```

Before and after each required element, there may be an arbitrary number lowercase letters, uppercase letters, and special characters. A union regular expression is needed for the lowercase letters, for the uppercase letters, for the special characters, and for the arbitrary characters that may appear between required characters. A union regular expression is used because it provides the ability to choose any element. These may be defined as follows:

```
(define LOWER (create-union-regexp lc))
(define UPPER (create-union-regexp uc))
(define SPCHS (create-union-regexp spc))
(define ARBTRY (kleenestar-regexp
                 (union-regexp LOWER
                               (union-regexp UPPER SPCHS))))
```

The creation of a chain of union regular expressions is delegated to an auxiliary function (to be designed and implemented). It is now possible to define a regular expression for each of the six orderings of required elements:

```
(define LUS (concat-regexp
              ARBTRY
              (concat-regexp
                LOWER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    UPPER
                    (concat-regexp
                      ARBTRY
                      (concat-regexp SPCHS ARBTRY)))))))
(define LSU (concat-regexp
              ARBTRY
              (concat-regexp
                LOWER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    SPCHS
                    (concat-regexp
                      ARBTRY
                      (concat-regexp UPPER ARBTRY)))))))
```

```
(define SLU (concat-regexp
              ARBTRY
              (concat-regexp
                SPCHS
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    LOWER
                    (concat-regexp
                      ARBTRY
                      (concat-regexp UPPER ARBTRY)))))))
(define SUL (concat-regexp
             ARBTRY
             (concat-regexp
               SPCHS
               (concat-regexp
                  ARBTRY
                 (concat-regexp
                  UPPER
                  (concat-regexp
                    ARBTRY
                    (concat-regexp LOWER ARBTRY)))))))

(define USL (concat-regexp
              ARBTRY
              (concat-regexp
                UPPER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    SPCHS
                    (concat-regexp
                      ARBTRY
                      (concat-regexp LOWER ARBTRY)))))))
(define ULS (concat-regexp
              ARBTRY
              (concat-regexp
                UPPER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    LOWER
                    (concat-regexp
                      ARBTRY
                      (concat-regexp SPCHS ARBTRY)))))))
```

The language of passwords is a word in any of the languages defined for the different orderings of required elements. It is defined using a union regular expression:

```
(define PASSWD (union-regexp
                LUS
                (union-regexp
                  LSU
                  (union-regexp
                    SLU
                    (union-regexp
                      SUL
                      (union-regexp USL ULS))))))
```

20.2 Design Idea

The constructor for a password takes no input and returns a string. A potential new password is locally defined. A word is generated by applying `FSM`'s `gen-regexp-word` to `PASSWD` and then converting the generated word to a string. If the length of the string is greater than or equal to 10, then it is returned as the generated password. Otherwise, a new word is generated.

Finally, in order to prevent generated passwords from getting unwieldy long, `gen-regexp-word` is given 5 as the maximum number of repetitions for a Kleene star regular expression. This value is arbitrary, and you may feel free to adjust it.

20.3 Function Definition

Following the steps of the design recipe yields the following function definition:

```
;;  → string
;; Purpose: Generate a valid password
(define (generate-password)
  (let [(new-passwd (passwd->string
                      (gen-regexp-word PASSWD 5)))]
    (if (>= (string-length new-passwd) 10)
        new-passwd
        (generate-password))))
```

We shall design and implement the auxiliary function, `passwd->string`, after completing the design of the function above.

20.4 Tests

Given that the function is nondeterministic, property-based testing is used. For this, a predicate that takes as input a string that may represent a password is needed. The given string is converted into a list of symbols. This list must have a length of at least 10 and contain at least one element from each of the following: `lowers`, `uppers`, and `spcls`. Following the steps of the design recipe yields:

```
;; string → Boolean
;; Purpose: Test if the given string is a valid password
(define (is-passwd? p)
  (let [(los (str->los p))]
    (and (>= (length los) 10)
         (ormap (λ (c) (member c los)) lowers)
         (ormap (λ (c) (member c los)) uppers)
         (ormap (λ (c) (member c los)) spcls))))
```

Converting the password to a list is delegated to an auxiliary function.

Sample tests using `check-pred` are:

```
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
```

These are not five repetitions of the same test because `generate-password` is nondeterministic.

20.5 Auxiliary Functions

Three auxiliary functions are needed: `create-union-regexp`, `str->los`, and `passwd->string`. The function `create-union-regexp` takes as input a list of regular expressions and returns a union regular expression. If the length of the given list is less than 2, an error is thrown because at least two regular expressions are needed for the union of regular expressions. If the given list only has two elements, then a union regular expression is constructed with the two regular expressions in the list. If the given list has a length greater than to 2, then a union regular expression is constructed with the first regular expression in the list and the union regular expression obtained from recursively processing the rest of the list. Following the steps of the design recipe yields:

```
;; (listof regexp) → union-regexp
;; Purpose: Create a union-regexp using the given list
;;          of regular expressions
(define (create-union-regexp L)
  (cond [(< (length L) 2)
         (error "create-union-regexp: list too short")]
        [(empty? (rest (rest L)))
         (union-regexp (first L) (second L))]
        [else
         (union-regexp (first L)
                       (create-union-regexp (rest L)))]))

;; Tests
(check-equal?
 (create-union-regexp (list (first lc) (first uc)))
 (union-regexp (singleton-regexp "a")
               (singleton-regexp "A")))
(check-equal?
 (create-union-regexp
  (list (first lc) (fourth uc) (third spc)))
 (union-regexp (singleton-regexp "a")
               (union-regexp (singleton-regexp "D")
                             (singleton-regexp "!"))))
```

The function `str->los` takes as input a string and returns a list of symbols. The given string is converted to a list of characters. Each character in the resulting list is converted to a symbol using `map`. The function given to `map` consumes a character and first converts the character into a string and then converts the string into a symbol. Following the steps of the design recipe yields:

```
;; string → (listof symbol)
;; Purpose: Convert the given string to a list of symbols
(define (str->los str)
  (map (λ (c) (string->symbol (string c)))
       (string->list str)))

;; Tests
(check-equal? (str->los "") '())
(check-equal? (str->los "a!Cop") '(a ! C o p))
```

Finally, the function `passwd->string` converts a given word representing a password into a string. First, the given word is traversed using `map`. The function given to `map` converts a symbol into a character by transforming the symbol into a string, transforming the string into a list of characters, and finally taking the first (and only) element in the list of characters. Second,

the list of characters produced by `map` is converted into a string. The steps of the design recipe produce:

```
;; word → string
;; Purpose: Convert the given password to a string
(define (passwd->string passwd)
  (list->string
   (map (λ (s)
          (first (string->list (symbol->string s))))
        passwd)))

;;Tests
(check-equal? (passwd->string '(a j h B ! ! y y t c))
              "ajhB!!yytc")
(check-equal? (passwd->string '($ u t q x ! J i n * K C))
              "$utqx!Jin*KC")
```

20.6 Running the Tests

Run the program and make sure all the tests pass. In addition, generate a few passwords. These are sample passwords generated:

```
> (generate-password)
"*j$xL!&CjMK"
> (generate-password)
"&$&*u&lBK&G*$&U!E!"
> (generate-password)
"!$*D!&$F!f!"
> (generate-password)
"$!*!eOY$!!*$$"
> (generate-password)
"$!bG&O&U&&!"
> (generate-password)
*xm!!**yIT"
```

The passwords generated appear fairly robust. It is unlikely that anyone would be able to guess any of them.

10 A DNA sequence may be represented as a word that contains the order in which an arbitrary number of the four bases appear. The bases are adenine (`A`), guanine (`G`), cytosine (`C`), and thymine (`T`). Define a regular expression for DNA sequences, and implement a function to generate a DNA sequence represented as a list.

11 Define a regular expression for real numbers, and implement a function to generate a real number.

12 A *user* is a random word generated from letters in the Roman alphabet and from the digits that starts with a letter. A *domain* is a random word generated from letters in the Roman alphabet followed by either `.com`, `.edu`, or `.net`. Consider the following definition for the language of email addresses:

$$\texttt{L = \{u@d | u is a user} \wedge \texttt{ d is a domain\}}$$

Define a regular expression for email addresses, and implement a function to generate an email address.

13 Let $\Sigma = \{\texttt{a b}\}$. Define a regular expression for Σ^*, and implement a function to generate a word in Σ^*.

14 Let $\Sigma = \{\texttt{a b}\}$. Define a regular expression, and implement a function to generate a word in the language for:

$$\texttt{L = \{w | w starts and ends with the same letter\}}$$

Chapter 5
Deterministic Finite-State Machines

Regular expressions define how to generate words in a regular language. Given a word, however, how can we decide if it is a member of a language? For this, it is desirable to have some type of device or machine that takes as input a word and returns `'accept` if the given word is in the language and `'reject` if the given word is not in the language. In essence, we need a model of a computer to determine if a word is part of a language.

How should such a machine operate? Analyzing how words in the language of a regular expression are generated can provide some insight. Consider how a word is generated for the following regular expression:

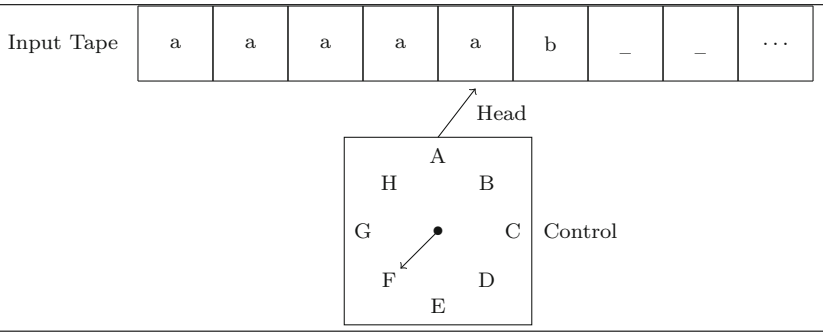
```
(concat-regexp
  (union-regexp (singleton-regexp "a")
                (singleton-regexp "b"))
  (concat-regexp (kleenestar-regexp (singleton-regexp "a"))
                 (singleton-regexp "b")))
```

Word generation traverses the structure of the regular expression. First, an `a` or a `b` is generated. Second, an arbitrary number of `a`s are generated. Third, a `b` is generated. If you think about it, the elements of a word are generated from left to right. This suggests that a word may be traversed from left to right to determine if it is in a language. We do not know if such a strategy will work. It is, nonetheless, plausible that it may work.

What should a machine that determines language membership look like? Clearly, it needs an input mechanism such as a tape where the input word is written. It needs a head to read an element from the tape and move right to the next symbol, if any, in the word. Finally, it needs a control mechanism that changes the state of the machine as the word is read. For instance, for the language of the regular expression above, the machine starts in a state where nothing has been read. After reading an `a` or `b`, it moves to state that

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_5

Fig. 15 A visualization of the finite-state automaton model



means that the initial symbol in the word matched what is expected for word membership.

Figure 15 displays a visual representation of the proposed machine. At the top, there is an input tape with the word `'(a a a a a b)`. The tape is of infinite length to the right. Each symbol in the word occupies one tape position. The placement of a word on the tape always starts with the tape's first position. At the bottom, there is a control module with eight states denoted by capital letters from `A` through `H`. In general, the number of states a machine may have is arbitrary. The arrow inside the control module indicates that state the machine is in. In Fig. 15, the machine is in state `F`. Finally, the head that reads from the tape is denoted by an arrow from the control module to an input tape position. The head is always over the next symbol to be read. Everything after the head has not been read. Everything before the head has been read. In Fig. 15, `'(a a a a)` has been read, `'(a b)` has not been read, and `a` is the next symbol to be read. The operational semantics of the machine is straightforward. Every time the head reads a symbol, it moves to the right. Based on a read symbol, the current state of the machine, and the transition rules of the machine, the control module moves to a (not necessarily different) state. The machine stops when the first blank (denoted by an underscore in Fig. 15) after the input is read. If the machine stops in a state that is an accepting state, then the machine accepts (i.e., the given word is in the language). Otherwise, it rejects (i.e., the given word is not in the language).

21 Deterministic Finite-State Machine Definition

To embark in the study of mathematical models of computers and algorithms, it is necessary to formally define machines. In this manner, we can precisely reason about them and write theorems about their properties. The machine

outlined above is called a *finite-state automaton* (or *finite-state machine*). It is a (very) restricted model of a computer. Like a modern computer, it has a `CPU` and can read input. It is restricted model in two important ways. The first is that it is only capable of accepting or rejecting words. That is, it is a language recognition machine. The second is that it has no memory other than what exists in the processing module. For example, it can remember the state that it is in but cannot remember the input read.

21.1 The `dfa` Constructor

We shall start our study of machine models by defining a *deterministic finite-state automaton*. A `dfa` is a type in `FSM` defined as follows:

```
A deterministic finite-state automaton, dfa, is a

    (make-dfa S Σ s F δ ['no-dead])
```

The inputs to the constructor are defined as follows:

S: A list of states. Each state is denoted by a symbol that represents a capital letter in the Roman alphabet.
Σ: A list of symbols or digits. Each symbol represents a lowercase letter in the Roman alphabet.
s: The starting state. It must be a member of `S`.
F: A list of final (i.e., accepting) states. Each state must be a member of `S`.
δ: A transition **function**. Each transition, $\delta(q, a) = r$, uniquely determines the state the machine moves to. Each transition is represented as a list with three elements: the from-state, the symbol read by the head, and the to-state. The two states must be in S, and the symbol read must be in Σ.

Given that δ is a function, it must contain a transition for every element in $S \times \Sigma$. In this regard, the constructor offers some flexibility. The constructor automatically adds a *dead state*, `ds` (denoted by the `FSM` constant `DEAD`), and any missing transitions. For any missing transition, the added transition moves the machine to the dead state. Transitions on the dead state go to the dead state. Sometimes, it is desirable (e.g., when you have written out the complete transition function or when visualizing a machine) to not add the dead state and the missing transitions. To inhibit the addition of the dead state, the optional argument `'no-dead` may be given to the constructor.

A computation for a `dfa`, `M`, is denoted by a list of configurations that `M` traverses to consume the input word. A configuration is a two-list that has the unconsumed part of the input word and the state of the machine. For instance, the configuration of the automaton in Fig. 15 is `((a b) F)`. A transition made (or step taken) by the machine is denoted using $\vdash$. $C_i \vdash C_j$

is valid for M if and only if M can move from C_i to C_j using a single transition. Zero or more moves by M is denoted using $\vdash^*$. $C_i \vdash^* C_j$ is valid for M if and only if M can move from C_i to C_j using zero or more transitions. A word, w, is accepted by M if the following is a valid computation:

```
(w s) ⊢ ^* ('() q), where q∈F
```

That is, M must entirely consume w and end in a final state. The language accepted by M, L(M), is the set of all strings accepted by M.

Consider the following dfa:

```
;; L(M) = ab*
(define M (make-dfa `(S F ,DEAD)
                    '(a b)
                    'S
                    '(F)
                    `((S a F)
                      (S b ,DEAD)
                      (F a ,DEAD)
                      (F b F)
                      (,DEAD a ,DEAD)
                      (,DEAD b ,DEAD))
                    'no-dead))
```

The language of this machine are all words that start with an a followed by an arbitrary number of bs. The addition of the dead state is suppressed because the transition function is fully specified (i.e., there is a transition rule for each member of (`(S F ,DEAD) × '(a b))). To determine if '(a b b b)∈L(M), we examine the computation performed:

```
((a b b b) S) ⊢ ((b b b) F) ⊢ ((b b) F) ⊢ ((b) F) ⊢ (() F)
```

Observe that '(a b b b) is entirely consumed and M ends in a final state. Therefore, '(a b b b)∈L(M).

21.2 FSM Machine Observers

FSM provides generic observers for all machine types. That is, these observers may be used with all state machines built with any FSM machine constructor. The observers for machine components are described as follows:

(sm-states m): Returns the states of the given machine
(sm-sigma m): Returns the alphabet of the given machine
(sm-rules m): Returns the transition relation of the given machine
(sm-start m): Returns the start state of the given machine
(sm-finals m): Returns the final states of the given machine
(sm-type m): Returns a symbol denoting the machine type

In addition, `FSM` provides two observers to apply a state machine to a word:

`(sm-apply m w [n])`: Applies the given machine to the given word. It returns either `'accept` or `'reject`. The optional natural number, `n`, is only used to indicate the starting position of the head for a Turing machine.

`(sm-showtransitions m w [n])`: Applies the given machine to the given word. It returns a list for the computation performed. The optional natural number, `n`, is only used to indicate the starting position of the head for a Turing machine.

To illustrate the use of the application observers, we use `M` defined above. Consider the following interactions:

```
> (sm-apply M '(a b b b))
'accept
> (sm-apply M '(a b b b a a b))
'reject
```

The first informs us that `'(a b b b)` is in `L(M)`. The second informs us that `'(a b b b a a b)` is not in `L(M)`. Let us look at the computation performed for each:

```
> (sm-showtransitions M '(a b b b))
'(((a b b b) S) ((b b b) F) ((b b) F) ((b) F) (() F) accept)
> (sm-showtransitions M '(a b b b a a b))
'(((a b b b a a b) S)
  ((b b b a a b) F)
  ((b b a a b) F)
  ((b a a b) F)
  ((a a b) F)
  ((a b) ds)
  ((b) ds)
  (() ds)
  reject)
```

The first is the computation developed by hand above. The second shows the computation performed to conclude that `'(a b b b a a b)`$\notin$`L(M)`. Observe that after reading the second `a`, the machine transitions to the dead state and remains there until the entire input word is consumed. The machine rejects because it halts in a non-final state.

21.3 `FSM` Machine Testers

In addition to the observers, `FSM` provides machine testers. The testers generate random words to test machines. The testers are:

`(sm-test m [n])`: Applies `m` to 100 randomly generated words and returns a list of the results. The optional natural number specifies the number of tests to perform.

`(sm-sameresult? m1 m2 w)`: Applies the two given machines, `m1` and `m2`, to, `w`, the given word and tests if the same result is obtained.

`(sm-testequiv? m1 m2 [n])`: Applies the 2 given machines, `m1` and `m2`, to the same 100 randomly generated words and tests if they produce the same results. If they do, true is returned. Otherwise, a list of the words for which the results differ is returned. The optional natural number specifies the number of words to test.

Using `M`, consider the following interaction that specifies testing ten words:

```
> (sm-test M 10)
'(((a a a a b b b) reject)
  ((a b) accept)
  (() reject)
  ((b b b a a) reject)
  ((a a b b a b a b b) reject)
  ((a b a a a) reject)
  ((a b a a b b) reject)
  ((b a b a a) reject)
  ((a b b b b) accept)
  ((a a b b a b b) reject))
```

Observe that only two words tested are in `L(M)`: `'(a b)` and `'(a b b b b)`. The other words are not in the language. A visual inspection of the results allows you to verify that the accepted words start with an `a` and then have an arbitrary number of `b`s. The rejected words do not. This ought to give you cautious optimism that `M` is correctly implemented.

To illustrate the use of `same-result?`, we define the following `dfa`:

```
;; L(M2) = abb*
(define M2 (make-dfa `(S A F ,DEAD)
                     '(a b)
                     'S
                     '(F)
                     `((S a A)
                       (S b ,DEAD)
                       (A a ,DEAD)
                       (A b F)
                       (F a ,DEAD)
                       (F b F)))
```

Consider the following interactions:

```
> (sm-sameresult? M M2 '(a b b))
#t
> (sm-sameresult? M M2 '(a))
#f
```

The first test informs us that `M` and `M2` yield the same result for `'(a b b)`. The second test informs us that `M` and `M2` yield different results for `'(a)` and, therefore, `L(M)` $\neq$ `L(M2)`.

To illustrate the use of `same-testequiv?`, consider the following interactions:

```
> (sm-testequiv? M M2)
'((a))
> (sm-testequiv? M M2 5)
'((a))
> (sm-testequiv? M M2 5)
#t
```

The first tests if `M` and `M2` produce the same result on 100 randomly generated words. The results inform us that for `'(a)`, the results differ. This makes sense because `'(a)`$\in$`L(M)` and `'(a)`$\notin$`L(M2)`. The second only performs tests using five randomly generated words and yields the same result. The third also only tests five randomly generated words but returns true suggesting that `L(M) = L(M2)`. How do the second and third interactions make sense? Recall that the tested words are randomly generated. The second interaction generated `'(a)`, while the third did not. This is why testing can only give us cautious optimism over what is being tested. If we want certainty, we need a proof.

21.4 `FSM` Machine Visualization

`FSM` provides machine rendering and machine execution visualization. The visualization primitives are:

`(sm-graph m)`: Returns a *transition diagram* rendered as a directed graph for the given machine.

`(sm-visualize m ) [(s p)*])` : Starts the `FSM` visualization tool for the given machine. The optional two-lists contain a state of the given machine and a predicate invariant (we will soon discuss this in more detail).

A machine's transition diagram is useful to visualize the states and the transition relation as a whole. For instance, for `M` (defined above), `(sm-graph M)` returns:

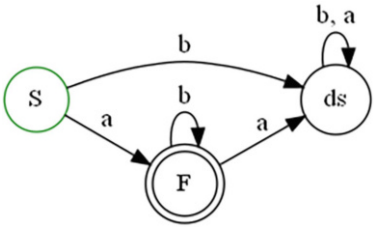
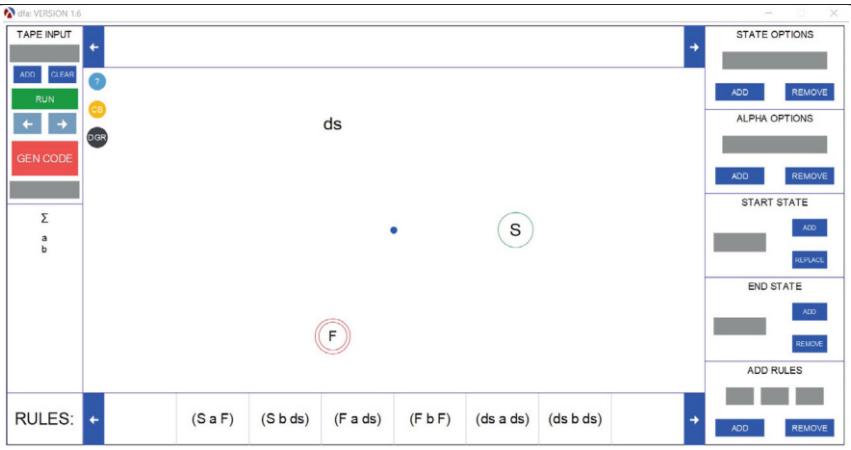
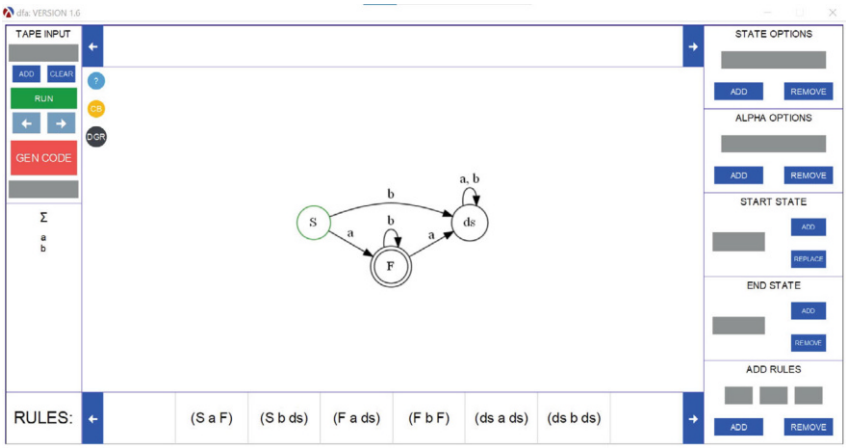


Fig. 16 Visualization of the `dfa` M



(a) Control View



(b) Graph View

The starting state is denoted in a green circle. Final states are denoted in double black circles. The arrows between states denote the transitions. The labels on the arrows denote the element read from the input tape. For example, `(S a F)` is denoted by the arrow from `S` to `F` with the label `a`. If there is more than one transition between any two states (not necessarily distinct), a single arrow is rendered with the consumed element for each transition separated by a comma. For instance, the transitions `(,DEAD a ,DEAD)` and `(,DEAD b ,DEAD)` are denoted by the edge from `ds` to `ds` with the label `b, a`.

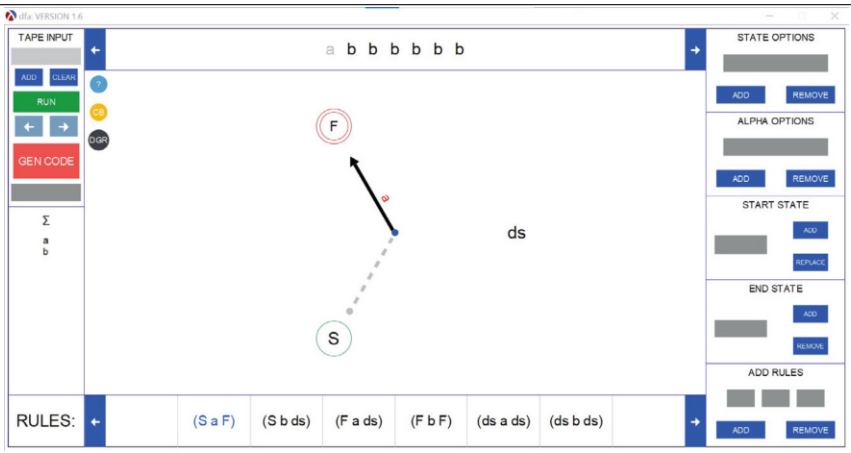
Typing `(sm-visualize M)` launches the `FSM` visualization tool. If `M` is successfully built, a pop-up window informs you of it, and you may close it. A machine may be visualized in two views, control view and graph view, as displayed in Fig. 16. The default is control view as displayed in Fig. 16a. In

either view, the right column allows for machine editing (e.g., adding states and transitions). If you edit the machine in the visualizer, you must hit the `Run` button in the left column. A pop-up window informs you if the edited machine was or was not successfully built. If successfully built, you may enter a word and simulate the execution of the machine. Otherwise, consult the interactions window for error messages. The left column allows you to enter an input word, run the machine, step through a computation, generate the `FSM` code for the machine visualized, and list the machine's alphabet(s). An input word is entered seven characters at a time separated by a space and clicking `ADD`. Longer words are entered adding the rest of the word in the same manner. To clear the input tape (and presumably enter a new word), click on the `CLEAR` button. Every time a word is entered, the `RUN` button must be clicked. The arrow buttons, ← and →, are used to step through a computation, respectively, backward and forward one transition at a time. To generate the code of a machine edited in the visualization tool, regardless of whether it builds or not, click on `GEN CODE`. A pop-up window informs you where the file is saved. At the top of the visualizer is the input tape, and at the bottom are the machine's transitions. In the center (for either view), there are three circular buttons. The top one takes you the `FSM` homepage, the middle button is to vary colors for those that suffer from some type of color blindness, and the bottom button is to switch between control view and graph view.

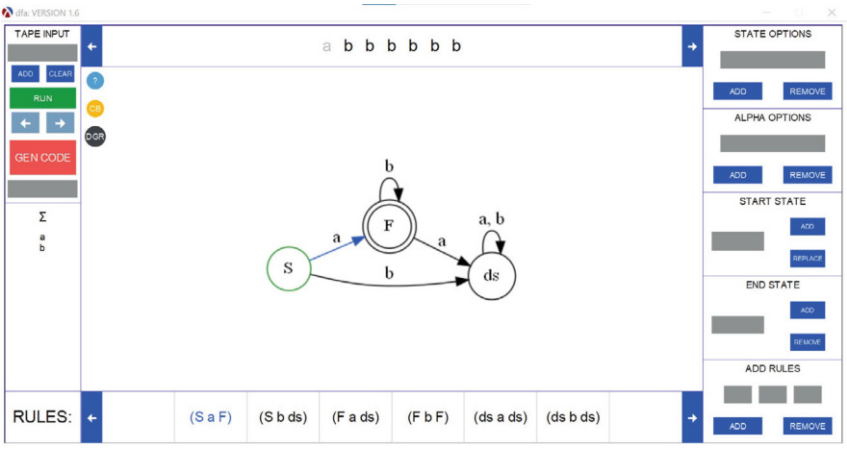
In control view, as displayed in Fig. 16a, the center contains the machine's states around a center point. The starting state is enclosed in a green circle, and final states are enclosed in double red circles. All other states are not encircled. When `RUN` is clicked, an arrow appears pointing to the starting state. Stepping through the computation fades out the consumed part of the word on the input tape, highlights the last rule used in the list of rules, draws a dashed line from the previous state to the center, and redraws the arrow to point to the current state. For instance, Fig. 17a displays the state of `M` after one transition. The machine is in state `F`; the previous state is `S`; the consumed input is `a`; the unconsumed input is `b b b b b b`; the last rule used, `(S a F)`, is highlighted; and the label on the arrow, `a`, is the last input symbol consumed.

In graph view, as displayed in Fig. 16b, the machine is visualized as rendered by `sm-graph`. Stepping through the computation also fades out the consumed part of the word on the input tape and highlights the last rule used in the list of rules. The arrow denoting the last rule used is highlighted in blue. For instance, Fig. 17b displays the state of `M` after one transition. The highlighted last transition moved the machine from `S` to `F` consuming an `a`. It is the same machine state as the one displayed in Fig. 17a.

Fig. 17 Visualization of the `dfa` M after one step



(a) Control view



(b) Graph view

22 A First Example

22.1 Designing the Machine

Assume Σ = {a b}. Consider implementing a `dfa` for the following language:

```
L = {w | w does not contain abaa}
```

We shall call our `dfa` `NO-ABAA` and its alphabet is, as the problem assumes, Σ = {a b}. The expected behavior of the machine is illustrated by the following tests:

```
(check-equal? (sm-apply NO-ABAA '()) 'accept)
(check-equal? (sm-apply NO-ABAA '(a a b a b b)) 'accept)
(check-equal?
  (sm-apply NO-ABAA '(a b b a a a b b b a b b a a b))
  'accept)
(check-equal? (sm-apply NO-ABAA '(a b a a))
              'reject)
(check-equal? (sm-apply NO-ABAA '(a b a b a a b a))
              'reject)
(check-equal?
  (sm-apply NO-ABAA '(a b a b b b a a b a a b b))
  'reject)
```

We need to define the conditions that must be tracked as an input word is consumed. In this example, we need to track how much of the prohibited pattern, `'(a b a a)`, has been detected. The pattern has length four. This means there are five conditions to track:

```
nothing detected
a    has been detected
ab   has been detected
aba  has been detected
abaa has been detected
```

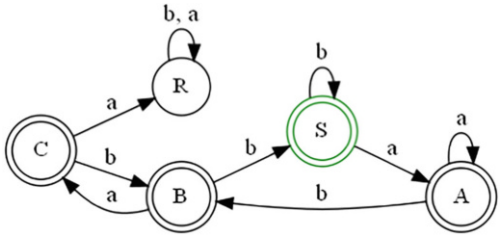
Each condition is associated with a state. The first condition must be the starting state because nothing in the prohibited pattern has been detected. The first four conditions define a final state because the prohibited pattern has not been detected. It is perfectly fine for the starting state to also be a final state. The final condition does not define a final state because it means the prohibited pattern is detected. The meaning of each state may be refined and documented in a program as follows:

```
;; L = {w | w does not contain abaa}
;; States
;;  S: nothing detected,       start and final state
;;  A: a    has been detected, final state
;;  B: ab   has been detected, final state
;;  C: aba  has been detected, final state
;;  R: abaa has been detected
```

A transition function needs to be defined. Given the current state of the machine and the symbol read from the input tape, it must move the machine to state that is consistent with the state definitions above. If the machine is in state `S` and an `a` is read, it moves to state `A` because this may be the first `a` in the prohibited pattern. If the machine is in state `S` and a `b` is read, it remains in state `S` because nothing in the prohibited pattern is detected. We may write the transition rules for `S` as follows:

Fig. 18 The `dfa` implementation and transition diagram for `NO-ABAA`

```
(define NO-ABAA
  (make-dfa
    '(S A B C R)
    '(a b)
    'S
    '(S A B C)
    '((S a A) (S b S)
      (A a A) (A b B)
      (B a C) (B b S)
      (C a R) (C b B)
      (R a R) (R b R))
    'no-dead))
```



(a) Machine implementation **(b)** Transition diagram

```
(S a A)
(S b S)
```

If the machine is in state `A` and an `a` is read, it remains in state `A` because the `a` read may be the first in the prohibited pattern. If the machine is in state `A` and a `b` is read, it moves to state `B` because `ab` has been detected. We may write the transition rules for `A` as follows:

```
(A a A)
(A b B)
```

If the machine is in state `B` and an `a` is read, it moves to state `C` because `aba` is detected. If the machine is in state `B` and a `b` is read, it moves to state `S` because none of the prohibited pattern is detected. We may write the transition rules for `B` as follows:

```
(B a C)
(B b S)
```

If the machine is in state `C` and an `a` is read, it moves to state `R` because `abaa` is detected. If the machine is in state `C` and a `b` is read, it moves to state `B` because the last two elements read, `ab`, may be the beginning of the prohibited pattern. We may write the transition rules for `C` as follows:

```
(C a R)
(C b B)
```

Finally, if the machine is in state `R`, then the input word contains the prohibited pattern, and the machine ought to remain in `R` as it reads the remaining part of the word. We may write the transition rules for `R` as follows:

```
(R a R)
(R b R)
```

The dfa implementation and transition diagram are displayed in Fig. 18. The constructor is signaled not to add a dead state because the transition function is fully specified. It is easy to visually verify that the transition function in Fig. 18a corresponds to the edges in the transition diagram in Fig. 18b. This machine can now be tested by running the unit tests and using sm-test. Once cautiously optimistic, after thorough testing, that the machine is correct, we must prove that L = L(NO-ABBA). To achieve this, first a predicate invariant is written in FSM for each state of NO-ABBA.

22.2 Writing dfa State Invariant Predicates

A dfa invariant predicate for a state, F, takes as input the consumed input, ci, and determines if ci satisfies the meaning of being in F. In addition, when developing a state invariant, always keep in mind that an invariant for a final state must establish that ci is in the language of the machine and for a non-final state must establish that ci is not in the language of the machine. To make writing state invariant predicates easier for NO-ABAA, define the prohibited pattern as follows:

```
(define PROHIBITED-PATTERN '(a b a a))
```

To start, let us reason about NO-ABAA's R state. According to our design, R means that the prohibited pattern has been detected. What does this mean in terms of ci? It means that ci has a minimum length of 4 and contains PROHIBITED-PATTERN. Observe that the second condition alone suffices to determine that the machine is in the correct state. Assuming the given word's length is at least 4, and following the steps of the design recipe, we arrive at the following state invariant predicate for R:

```
;; word → Boolean
;; Purpose: Determine if the consumed input contains
;;          PROHIBITED-PATTERN
;;  Assume: |ci| >= 4
(define (R-INV ci) (contains? ci PROHIBITED-PATTERN))

;; Tests for R-INV
(check-equal? (R-INV '(a)) #f)
(check-equal? (R-INV '(a b)) #f)
(check-equal? (R-INV '(a b a)) #f)
(check-equal? (R-INV '(a b a a)) #t)
(check-equal? (R-INV '(a a b b a b a a b b a)) #t)
```

The job of determining if the consumed input contains the prohibited pattern is delegated to an auxiliary function. This auxiliary predicate traverses the given word searching for the given pattern. It takes as input two words, w

and `pattern`, and returns a Boolean. There are three conditions it must distinguish. If the length of `w` is less than the length of `pattern`, it returns `#f` because `w` does not contain `pattern`. If the first |`pattern`| symbols in `w` equal `pattern`, then it returns `#t` because `w` contains `pattern`. Otherwise, it recursively determines if the rest of `w` contains `pattern`. Following the steps of the design recipe yields:

```
;; word word → Boolean
;; Purpose: Determine if the second given word appears in
;;          the first given word
(define (contains? w pattern)
  (cond [(< (length w) (length pattern)) #f]
        [(equal? (take w (length pattern)) pattern) #t]
        [else (contains? (rest w) pattern)]))

;; Tests for contains?
(check-equal? (contains? '() PROHIBITED-PATTERN) #f)
(check-equal? (contains? '(a b b a a) PROHIBITED-PATTERN)
              #f)
(check-equal? (contains? '(b b b a b a b b a b a)
                         PROHIBITED-PATTERN)
              #f)
(check-equal? (contains? '(a b a a)
                         PROHIBITED-PATTERN)
              #t)
(check-equal? (contains? '(a b b b a b a a a)
                         PROHIBITED-PATTERN)
              #t)
(check-equal? (contains? '(a b a b a a a b a a b)
                         PROHIBITED-PATTERN)
              #t)
```

The function `take` requires as input a list, `L`, and a natural number, `n`, and returns a list with the first `n` elements of `L`.

What must be true about `ci` if `NO-ABAA` is in state `C`? From our design, we know that the `ci` must end with `'(a b a)` and that `C` is a final state. This means that `ci`'s minimum length is 3 and that `ci` does not contain the prohibited pattern. Observe that these conditions suffice to establish that the machine is in the correct state. `C`'s invariant predicate is written assuming `ci`'s minimum length is 3. It determines if `ci` ends with `'(a b a)` and if it does not contain the prohibited pattern. Following the steps of the design recipe yields:

```
;; word → Boolean
;; Purpose: Determine if the given word ends with aba
;;          and does not contain PROHIBITED-PATTERN
;;  Assume: |word| >= 3
(define (C-INV ci)
  (and (equal? (drop ci (- (length ci) 3)) '(a b a))
       (not (contains? ci PROHIBITED-PATTERN))))

;; Tests for C-INV
(check-equal? (C-INV '(a b a)) #t)
(check-equal? (C-INV '(a b a b b a a b a b a)) #t)
(check-equal? (C-INV '(a b b)) #f)
(check-equal? (C-INV '(a b b a b a a b a)) #f)
```

Observe that, once again, the assumption is made explicit. This helps any readers of the code understand its design. For example, it explains why the unit tests do not test words of length less than 3. The function `drop` takes as input a list, `L`, and a natural number, `n`, and returns a list that is `L` after removing `L`'s first `n` elements.

What does it mean for the machine to be in state `B`? Our design states that `B` is a final state and that `'(a b)` has been detected. This means that `ci`'s minimum length is 2. In addition, `ci` must end with `(a b)` and cannot contain the prohibited pattern. These conditions suffice to establish that the machine is in the correct state. Following the steps of the design recipe leads to:

```
;; word → Boolean
;; Purpose: Determine if the given word ends with ab
;;          and does not contain PROHIBITED-PATTERN
;;  Assume: |word| >= 2
(define (B-INV ci)
  (and (equal? (drop ci (- (length ci) 2)) '(a b))
       (not (contains? ci PROHIBITED-PATTERN))))

;; Tests for B-INV
(check-equal? (B-INV '(a b)) #t)
(check-equal? (B-INV '(a a b)) #t)
(check-equal? (B-INV '(a b b b a b a b)) #t)
(check-equal? (B-INV '(a a b a a)) #f)
(check-equal? (B-INV '(a b a)) #f)
(check-equal? (B-INV '(a b a b a b a)) #f)
(check-equal? (B-INV '(a b b b a b a a b b a)) #f)
```

What does it mean for the machine to be in state `A`? Based on our design, `ci` must end with `a` and not contain the prohibited pattern. Note that these

conditions do not suffice to establish that the machine is in the correct state. This follows from observing that both conditions are true for any consumed input that takes the machine to `C`. Therefore, this invariant predicate must also guarantee that `ci` does not end with `'(a b a)`. How can `ci` not ending with `'(a b a)` be determined? This is determined by testing if $|$`ci`$|$ `<` `3` or if $|$`ci`$| \geq$ `3` and `ci` does not end with `'(a b a)`. The steps of the design recipe lead to:

```
;; word → Boolean
;; Purpose: Determine if the consumed input
;;          ends with a, does not contain
;;          the prohibited input, and does
;;          not end with aba.
;; Assume: |word| > 0
(define (A-INV ci)
  (and (equal? (drop ci (sub1 (length ci))) '(a))
       (not (contains? ci PROHIBITED-PATTERN))
       (or (< (length ci) 3)
           (not (equal? (drop ci (- (length ci) 3))
                        '(a b a))))))

;; Tests for A-INV
(check-equal? (A-INV '(b)) #f)
(check-equal? (A-INV '(a b)) #f)
(check-equal? (A-INV '(a b a)) #f)
(check-equal? (A-INV '(a b a a b a a b)) #f)
(check-equal? (A-INV '(a)) #t)
(check-equal? (A-INV '(a a)) #t)
(check-equal? (A-INV '(b b a)) #t)
(check-equal? (A-INV '(a b b a b a b b a)) #t)
```

The second branch of the `or`-expression is only evaluated if the expression in the first branch is `#f`. That is, it is only evaluated if $|$`ci`$| \geq 3$.

Finally, what does it mean for the machine to be in state `S`? Our design informs us that `ci` cannot end with a nonempty prefix of the prohibited pattern[7] (i.e., none of the prohibited pattern is detected) and that it is the start and a final state. We can conclude that machine is in the correct state if `ci = '()`. What if `ci` $\neq$ `'()`? In this case, `ci` cannot contain the prohibited pattern and must end with a `b`. Observe that these conditions do not suffice to establish that the machine is in the correct state because they also hold for any consumed input that takes the machine to state `B`. Therefore, this invariant predicate must also establish that `ci` does not end with `'(a b)`. How is this determined? It is determined by testing if $|$`ci`$| = 1$ or if $|$`ci`$| > 1$ and `ci` does not end in `(a b)`. These observations and the steps of the design recipe produce:

[7] Recall that w is a prefix of w. Do not confuse prefix with proper prefix.

```
;; word → Boolean
;; Purpose: Determine if NO-ABAA should be in S
(define (S-INV ci)
  (or (= (length ci) 0)
      (and (not (contains? ci PROHIBITED-PATTERN))
           (eq? (last ci) 'b)
           (or (= (length ci) 1)
               (not (equal? (drop ci (- (length ci) 2))
                            '(a b)))))))

;; Tests for S-INV
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(b)) #t)
(check-equal? (S-INV '(b b)) #t)
(check-equal? (S-INV '(a b b)) #t)
(check-equal? (S-INV '(b a b b a b a b b)) #t)
(check-equal? (S-INV '(a)) #f)
(check-equal? (S-INV '(a b)) #f)
(check-equal? (S-INV '(a b a)) #f)
(check-equal? (S-INV '(a b a a)) #f)
(check-equal? (S-INV '(a b b b b a b a a b b)) #f)
(check-equal? (S-INV '(a a b b b b a b a b b a b a)) #f)
```

Once state invariant predicates are written, validate them by running the unit tests and visualizing machine execution. The syntax to visualize `NO-ABAA` with its invariants is:

```
(sm-visualize NO-ABAA
              (list 'S S-INV) (list 'A A-INV)
              (list 'B B-INV) (list 'C C-INV)
              (list 'R R-INV))
```

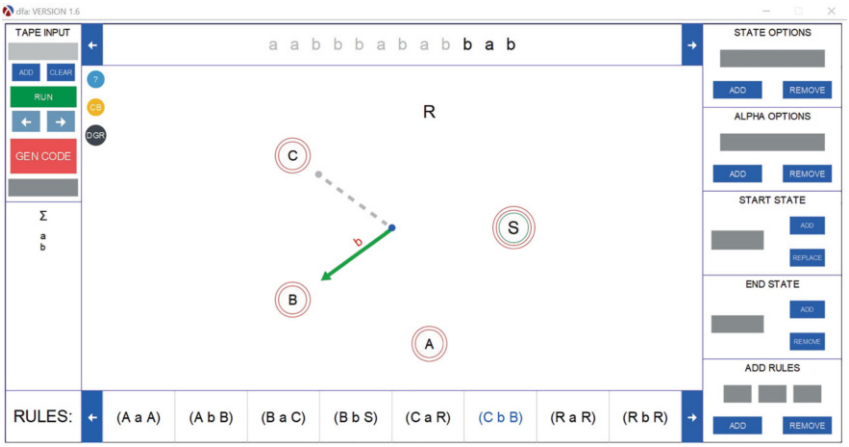
Figure 19 displays a visualization image for `NO-ABAA` with invariants in control view. You can observe that the arrow is green. This means that the invariant for the machine's current state holds. If the invariant does not hold, then the arrow turns red. If a state invariant has a bug and returns a value that is not a Boolean, the arrow turns yellow. In state diagram view, the states are, respectively, shaded green, red, and yellow.

You can step backward and forward through a computation to validate your state invariant predicates. Do so for several different inputs. This is important because it may help you discover bugs in your machine or in your state invariants. If there is a bug, of course, it needs to be corrected before attempting to prove that the machine is correct.

It is also important to note that not all invariants need to be provided to the visualizer. This allows for piecemeal development and validation of state invariant predicates. For instance, run the visualizer this way:

```
(sm-visualize NO-ABAA (S S-INV) (R R-INV))
```

Fig. 19 Visualizing `NO-ABAA` with invariant predicates



For `S` and `R`, the arrow will change color and remain black for all other states.

22.3 Proving `L(NO-ABAA) = L`

It is not enough to design a machine and validate its state invariants. We must prove that the machine correctly decides language membership for an arbitrary word. Assume that a `dfa`, `M`, is designed for a language `L`. The proof that `L = L(M)` is done in two steps for an arbitrary word, `w`, such that `w∈(sm-sigma M)`*:

1. Prove the state invariants hold when M is applied to w.
2. Prove that if invariants hold when M is applied to w, then L = L(M).

The first is done by induction on the number of transitions performed by `M`. The second uses the invariants to prove:

$$\texttt{w}\in\texttt{L} \Leftrightarrow \texttt{w}\in\texttt{L(M)} \wedge \texttt{w}\notin\texttt{L} \Leftrightarrow \texttt{w}\notin\texttt{L(M)}$$

Let us prove that `L(NO-ABAA)=L`. Let `M = NO-ABAA`, `S=(sm-states M)`, `s=(sm-start M)`, Σ`=(sm-sigma M)`, `w`$\in \Sigma^*$, and `ci` = the consumed input (i.e., the consumed part of `w`).

22.3.1 Proving Invariants Hold for `NO-ABAA`

Theorem 1 *State invariants hold when NO-ABAA is applied to w.*

Proof
Proof by induction on n = the number of steps M performs to consume w.

Base Case: n = 0
If n is 0, then the consumed input must be `'()`. Clearly, `S-INV` holds because `(= (length ci) 0)`.
Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.

If n=k+1, then the consumed input cannot be `'()` given that the machine must have consumed at least one symbol. Therefore, we can state that ci=xa such that |ci|=k+1, x$\in\Sigma^*$ and a$\in\Sigma$. M's computation to consume ci has k+1 steps:

$$(\text{ci s}) = (\text{xa s}) \vdash^k (\text{a r}) \vdash \ (\texttt{'()}\ \ \text{q}), \text{ where r,q}\in S \qquad \square$$

That is, M consumes x in k steps and reaches some state r. Then in one step, it consumes `a` to reach q. Given that |x|=k, the inductive hypothesis informs us that the state invariants hold when x is consumed by M. We must show that the state invariants hold for the k+1 transition into q.

Consider every transition independently:

`(S a A)`: By inductive hypothesis, S-INV holds. Consuming `a` means that:

- The consumed input ends with `a`.
- The consumed input does not contain the prohibited pattern because S-INV guarantees the consumed input before the consumed `a` does not contain the prohibited pattern and ends with a `b`. This means that after consuming `a`, the consumed input does not end with the prohibited pattern. Therefore, the consumed input (including the newly consumed `a`) does not contain the prohibited pattern.
- The consumed input does not end with `'(a b a)` for the following reasons:
  - If the length of the consumed input is less than 3, then it does not end with `'(a b a)`.
  - If the length of the consumed input is greater than or equal to 3, then it does not end with `'(a b a)` because S-INV guarantees that the consumed input before the `a` does not end with `'(a b)`.

Therefore, we may conclude that A-INV holds after the machine consumes the `a`.

`(S b S)`: By inductive hypothesis, S-INV holds. Consuming `b` means that:

- The consumed input ends with `b`.
- The consumed input does not contain the prohibited pattern because the S-INV guarantees that anything before the consumed `b` does not contain the prohibited pattern and the prohibited pattern does not end with `b`.
- If the consumed input before the consumed `b` is `'()`, then the length of the consumed input is 1 and does not end with `'(a b)`.
- If the consumed input before the consumed `b` is not `'()`, then the consumed input does not end with `'(a b)` because the S-INV guarantees that the consumed input before consuming `b` ends with `b`.

Therefore, we can conclude that S-INV holds after the machine consumes the `b`.

`(A a A)`: By inductive hypothesis, A-INV holds. Consuming `a` means that:

- The consumed input ends with `a`.
- The consumed input does not contain the prohibited pattern because A-INV guarantees that the consumed input before the consumed `a` does not contain the prohibited pattern and does not end with `'(a b a)`.
- If the length of the consumed input is less than 3, then it does not end with `'(a b a)`.
- If the length of the consumed input is greater than or equal to 3, then it does not end with `'(a b a)` because A-INV guarantees that the input before the consumed `a` ends with `a`.

Therefore, we may conclude that A-INV holds when the machine consumes the `a`.

`(A b B)`: By inductive hypothesis, A-INV holds. Consuming `b` means that:

- The consumed input ends with `'(a b)` because A-INV guarantees that the consumed input before the consumed `b` ends with `a`.
- The consumed input does not contain the prohibited pattern because A-INV guarantees that the consumed input before the consumed `b` does not contain the prohibited pattern and the prohibited pattern does not end with `b`.

Therefore, we may conclude that B-INV holds when the machine consumes the `b`.

`(B a C)`: By inductive hypothesis, B-INV holds. Consuming `a` means that:

- The consumed input ends with `'(a b a)` because B-INV guarantees that the consumed input before the consumed `a` ends with `'(a b)`.
- B-INV guarantees that the consumed input before the consumed `a` does not contain the prohibited pattern and ends with `'(a b)`. This means

that the consumed input does not contain the prohibited pattern because it is not contained before the `a` consumed and the prohibited pattern does not end with `'(a b a)`.

Therefore, we may conclude that C-INV holds when the machine consumes the `b`.

(B b S): By inductive hypothesis, B-INV holds. Consuming `b` means that:

- The consumed input ends with `b`.
- B-INV guarantees that the consumed input before the consumed `b` does not contain the prohibited pattern. Consuming `b` means that the consumed input does not end with the prohibited pattern. Therefore, the consumed input does not contain the prohibited pattern.
- B-INV guarantees that the consumed input ends with `'(a b)`. This means that after consuming `b`, the consumed input does not end with `'(a b)`.

Therefore, we may conclude that S-INV holds when the machine consumes the `b`.

`(C a R)`: By inductive hypothesis, C-INV holds. Consuming `a` means that the consumed input ends with `'(a b a a)` because C-INV guarantees that the consumed input before the consumed `a` ends with `'(a b a)`. Observe that `'(a b a)` and `a` form the prohibited pattern. Therefore, we may conclude that R-INV holds when the machine consumes `a`.

`(C b B)`: By inductive hypothesis, C-INV holds. Consuming `b` means that:

- The consumed input ends with `'(a b)` because C-INV guarantees that the consumed input before consuming `b` ends with `'(a b a)`.
- The consumed input does not contain the prohibited pattern because C-INV guarantees that the consumed input before the consuming `b` does not contain the prohibited pattern and the prohibited pattern does not end with `b`. □

Therefore, we may conclude that B-INV holds when the machine consumes the `b`.

`(R a R)`: By inductive hypothesis, R-INV holds. R-INV guarantees that the consumed input before consuming `a` contains the prohibited pattern. Consuming `a` means that the consumed input still contains the prohibited pattern. Therefore, we may conclude that R-INV holds when the machine consumes `a`.

`(R b R)`: By inductive hypothesis, R-INV holds. R-INV guarantees that the consumed input before the consumed `b` contains the prohibited pattern. Consuming `b` means that the consumed input still contains the prohibited pattern. Therefore, we may conclude that R-INV holds when the machine consumes `b`.

22.3.2 Proving `L(NO-ABAA) = L`

The proof that `L(NO-ABAA) = L` is divided into two pieces called lemmas:

1. w∈L ⇔ w∈L(M)
2. w∉L ⇔ w∉L(M)

Lemma 1 *w∈L ⇔ w∈L(M)*

Proof
(⇒) Assume w∈L.

w∈L means that w does not contain `'(a b a a)`. Given that state invariants always hold, M cannot consume w and halt in R. This means that M must halt in S, A, B, or C. These are final states and, therefore, w∈L(M).

(⇐) Assume w∈L(M).

w∈L(M) means that M halts in S, A, B, or C. The invariants for these states inform us that w does not contain `'(a b a a)`. Therefore, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(M)*

Proof
(⇒) Assume w∉L.

w∉L means that w contains `'(a b a a)`. The state invariants inform us that M must halt in R when it consumes w. R is not a final state of M. Therefore, w∉L(M).

(⇐) Assume w∉L(M).

The state invariants inform us that M halts in R when M consumes w. This means that w contains `'(a b a a)`. Therefore, w∉L. □

Theorem 2 *L(NO-ABAA) = L*

Proof
Lemmas 1 and 2 establish the theorem. □

23 A Design Recipe for State Machines

Based on `NO-ABAA`'s development, a design recipe for state machines is outlined in Fig. 20. The design recipe presents guiding steps to follow for the development of state machines (including state machines not yet discussed).

Fig. 20 Design recipe for state machines

To design and implement a state machine, M, for a language L:

1. Name the machine and specify alphabets
2. Write unit tests
3. Identify conditions that must be tracked as input is consumed, associate a state with each condition, and determine the start and final states.
4. Formulate the transition relation
5. Implement the machine
6. Test the machine using unit tests and random testing
7. Design, implement, and test an invariant predicate for each state
8. Prove L = L(M)

Step 1 asks you to pick a descriptive name for the machine and to define the needed alphabets (e.g., a `dfa`'s input alphabet).

Step 2 asks for the development of unit tests. Write tests for words that are rejected and for words that are accepted. The tests ought to be thorough. Testing a comprehensive set of words is important to make sure all possible characteristics of word in the language and words not in the language are correctly processed. Remember that thorough testing leads to cautious optimism about the correctness of your design and implementation. In addition, it helps readers of your code understand the machine's expected behavior.

Step 3 asks for the conditions that must be tracked as the machine consumes a word. Associate each condition with a state. Each condition describes properties of the consumed input. Be as specific as possible because it will make other steps in the design recipe easier. Clearly annotate the start and final/accepting states.

Step 4, based on the result of step 3, asks for the development of the transition relation. In the case of a `dfa`, for example, this relation must be a function. Develop each transition assuming that the condition describing the source state holds and that the action(s) taken by the machine make the conditions of the destination state hold. For instance, consider developing a transition for a `dfa`. Given a state, `A`, and an input alphabet, `a`, assume that the conditions `A` represents hold and formulate the conditions that hold after consuming `a`. If the conditions that hold after consuming `a` are state `B`'s conditions, then the needed transition is `(A a B)`. If the conditions that hold do not satisfy the conditions of any state, then you either discovered the need for a new state, or the conditions represented by a state need to be refined.

Steps 5 and 6 ask for the machine's implementation and the running of unit and random tests. The machine is implemented by defining a variable for the name selected in step 1 and using an `FSM` machine constructor (e.g., `make-dfa`). Random testing may be done using `sm-test`.

Step 7 asks for the development of state invariant predicates. A state invariant predicate takes as input the varying elements, like the consumed input, and verifies that the conditions the state represents hold.

Step 8 asks for the development of a proof demonstrating that the machine's language, `L(M)`, is the same as the language, `L`, the machine is designed to decide. This is done by first proving by induction that the state invariants hold when an arbitrary word is processed. This proof is then used to prove that `L = L(M)`.

24 The State Machine Design Recipe in Action

To illustrate the machine design recipe in action, consider designing and implementing a `dfa` for the following language:

$$\texttt{L} = \{\texttt{w} \mid \texttt{w} \in \{\texttt{a b}\}^* \wedge \texttt{w has an even number of a and an odd number of b}\}$$

`L` contains all words made of an arbitrary number of `a`s and `b`s such that the number of `a`s is even and the number of `b`s is odd. The following subsections outline the results for each step of the design recipe.

24.1 Name and Alphabet

Every word must have an even number of `a`s and `b`s and must only contain `a`s and `b`s. The machine name and input alphabet are defined as:

```
;; Name: EVEN-A-ODD-B
;;
;; Σ: '(a b)
```

24.2 Unit Tests

The tests cover words of different lengths such that some have an even number of `a`s and an odd number of `b`s and some do not:

```
;; Tests for EVEN-A-ODD-B
(check-equal? (sm-apply EVEN-A-ODD-B '()) 'reject)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b a)) 'reject)
(check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a))
              'reject)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b)) 'reject)
(check-equal? (sm-apply EVEN-A-ODD-B '(a b b b b))
              'reject)
```

```
(check-equal? (sm-apply EVEN-A-ODD-B '(b a b b a a b))
              'reject)
(check-equal? (sm-apply EVEN-A-ODD-B '(b)) 'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(a a b)) 'accept)
(check-equal? (sm-apply EVEN-A-ODD-B '(a a a b a b b))
              'accept)
```

24.3 States

As a word is processed, the consumed input may contain:

1. An even number of a and an even number of b
2. An odd number of a and an odd number of b
3. An even number of a and an odd number of b
4. An odd number of a and an even number of b

When processing starts, the consumed input has 0 as and 0 bs. That is, the consumed input has an even number of as and an even number of bs. This means that the state that captures this condition must be the starting state. The state that represents that the consumed input has an even number of as and an odd number of bs must be the only final state. The states may be documented as follows:

```
;; States
;; S: even number of a and even number of b, start state
;; M: odd number of a and odd number of b
;; N: even number of a and odd number of b, final state
;; P: odd number of a and even number of b
```

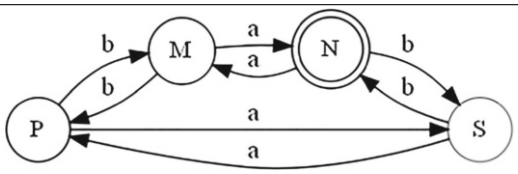
24.4 The Transition Function

If the consumed input has an even number of as and an even number of bs, then after consuming an a, the consumed input has an odd number of as and an even number of bs, and after consuming a b, the consumed input has an even number of as and an odd number of bs. This means that the following are needed transitions in EVEN-A-ODD-B:

```
(S a P)
(S b N)
```

If the consumed input has an odd number of as and an odd number of bs, then after consuming an a, the consumed input has an even number of as and an odd number of bs, and after consuming a b, the consumed input

Fig. 21 EVEN-A-ODD-B's transition diagram



has an odd number of as and an even number of bs. This means that the following are needed transitions in EVEN-A-ODD-B:

```
(M a N)
(M b P)
```

If the consumed input has an even number of as and an odd number of bs, then after consuming an a, the consumed input has an odd number of as and an odd number of bs, and after consuming a b, the consumed input has an even number of as and an even number of bs. This means that the following are needed transitions in EVEN-A-ODD-B:

```
(N a M)
(N b S)
```

If the consumed input has an odd number of as and an even number of bs, then after consuming an a, the consumed input has an even number of as and an even number of bs, and after consuming a b, the consumed input has an odd number of as and an odd number of bs. This means that the following are needed transitions in EVEN-A-ODD-B:

```
(P a S)
(P b M)
```

Observe that for every state, there is exactly one transition for every alphabet member. Thus, a transition function is fully defined.

24.5 Implementation and Testing

EVEN-A-ODD-B is implemented as follows:

```
(define EVEN-A-ODD-B (make-dfa '(S M N P)
                               '(a b)
                               'S
                               '(N)
                               '((S a P)
                                 (S b N)
                                 (M a N)
```

```
    (M b P)
    (N a M)
    (N b S)
    (P a S)
    (P b M))
  'no-dead))
```

The `EVEN-A-ODD-B`'s transition diagram is displayed in Fig. 21.

Running the program results in all unit tests passing. To further test `EVEN-A-ODD-B`, use `sm-test`. A sample of random tests is:

```
> (sm-test EVEN-A-ODD-B 20)
'(((b a a a) reject)
  ((a a b a a b b) accept)
  ((b b a) reject)
  ((a b a) accept)
  (() reject)
  ((a a a a) reject)
  ((b b b a a b b) accept)
  ((b b b a b a b) accept)
  ((b a b a a b b b) reject)
  ((b b b b) reject)
  ((a b) reject)
  ((a b b b b b a) accept)
  ((b a a a b) reject)
  ((b a a a a) accept)
  ((b b b a b a a) reject)
  ((b b b a b) reject)
  ((a a) reject)
  ((a b b b a a b b) reject)
  ((a) reject)
  ((b a b b a a a b b) accept))
```

A visual inspection of the results confirms that all test results are correct.

24.6 State Invariant Predicates

A state invariant must establish that the conditions the state represents hold and that the machine should not be in any other state. For a `dfa`, a state invariant takes in the consumed input and returns a Boolean.

For `S`, the state invariant predicate must establish that the consumed input has an even number of `a`s and an even number of `b`s. Observe this suffices to establish that the machine should not be in any other state. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if given word has an even number
;;          of a and an even number of b
(define (S-INV ci)
  (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))
       (even? (length (filter (λ (s) (eq? s 'b)) ci)))))

;; Tests for S-INV
(check-equal? (S-INV '(a)) #f)
(check-equal? (S-INV '(a b b b a)) #f)
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(a a b b)) #t)
```

For `M`, the state invariant predicate must establish that the consumed input has an odd number of `a`s and an odd number of `b`s. Observe this suffices to establish that the machine should not be in any other state. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if given word has an odd number
;;          of a and an odd number of b
(define (M-INV ci)
  (and (odd? (length (filter (λ (s) (eq? s 'a)) ci)))
       (odd? (length (filter (λ (s) (eq? s 'b)) ci)))))

;; Tests for M-INV
(check-equal? (M-INV '(a)) #f)
(check-equal? (M-INV '(a b b b a)) #f)
(check-equal? (M-INV '(a b b b a a b)) #f)
(check-equal? (M-INV '(b a)) #t)
(check-equal? (M-INV '(b a a b a b)) #t)
```

For `N`, the state invariant predicate must establish that the consumed input has an even number of `a`s and an odd number of `b`s. Observe this suffices to establish that the machine should not be in any other state. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if given word has an even number
;;          of a and an odd number of b
(define (N-INV ci)
  (and (even? (length (filter (λ (s) (eq? s 'a)) ci)))
       (odd? (length (filter (λ (s) (eq? s 'b)) ci)))))

;; Tests for N-INV
(check-equal? (N-INV '()) #f)
(check-equal? (N-INV '(a b a b a)) #f)
```

```
(check-equal? (N-INV '(a b b a a b)) #f)
(check-equal? (N-INV '(b a a)) #t)
(check-equal? (N-INV '(a b a a b a b b b)) #t)
```

For P, the state invariant predicate must establish that the consumed input has an odd number of as and an even number of bs. Observe this suffices to establish that the machine should not be in any other state. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if given word has an odd number
;;          of a and an even number of b
(define (P-INV ci)
  (and (odd? (length (filter (λ (s) (eq? s 'a)) ci)))
       (even? (length (filter (λ (s) (eq? s 'b)) ci)))))

;; Tests for P-INV
(check-equal? (P-INV '()) #f)
(check-equal? (P-INV '(a b)) #f)
(check-equal? (P-INV '(a b b a a b a)) #f)
(check-equal? (P-INV '(b a b)) #t)
(check-equal? (P-INV '(a b a a b b b)) #t)
```

24.7 Correctness Proof

For the required proofs, we use the following notation:

```
 M = EVEN-A-ODD-B
Σ = (sm-sigma M)
 F = (sm-finals M)
 w ∈ Σ*
ci = the consumed input
```

24.7.1 Proof That Invariants Hold

Theorem 3 *The state invariants hold when M is applied to w.*

Proof
Proof by induction on the number of transitions, n, M makes to consume w.

Base Case: n = 0
If n is 0, then the consumed input is '(), and M is in S. This means the consumed input has an even number of as and an even number of bs (0 of each). Therefore, S-INV holds.

Inductive Step:
Assume: State invariants hold for n = k.
Show: State invariants hold for n = k+1.

If n=k+1, then the consumed input cannot be `'()` given that the machine must have consumed at least one symbol. Therefore, we can state that ci=xa such that |ci|=k+1, x$\in\Sigma^*$ and a$\in\Sigma$. M's computation to consume ci has k+1 steps:

$$(\text{xa s}) \vdash^{k} (\text{a r}) \vdash \ (\texttt{'()} \ \ \text{q}), \text{ where r,q}\in\text{S} \qquad \square$$

Given that |x|=k, the inductive hypothesis informs us that the state invariants hold when x is consumed by M. We must show that the state invariants hold for the k+1 transition into q. That is, we must show that for every transition, the invariant holds for the state transitioned into. Consider every transition independently:

`(S a P)`: Assume S-INV holds. Consuming an `a` means ci has an odd number of `a`s and an even number `b`s. Therefore, P-INV holds.

`(S b N)`: Assume S-INV holds. Consuming a `b` means ci has an even number of `a`s and an odd number `b`s. Therefore, N-INV holds.

`(M a N)`: Assume M-INV holds. Consuming an `a` means ci has an even number of `a`s and an odd number `b`s. Therefore, N-INV holds.

`(M b P)`: Assume M-INV holds. Consuming an `b` means ci has an odd number of `a`s and an even number `b`s. Therefore, P-INV holds.

`(N a M)`: Assume N-INV holds. Consuming an `a` means ci has an odd number of `a`s and an odd number `b`s. Therefore, M-INV holds.

`(N b S)`: Assume N-INV holds. Consuming an `b` means ci has an even number of `a`s and an even number `b`s. Therefore, S-INV holds.

`(P a S)`: Assume P-INV holds. Consuming an `a` means ci has an even number of `a`s and an even number `b`s. Therefore, S-INV holds.

`(P b M)`: Assume P-INV holds. Consuming an `b` means ci has an odd number of `a`s and an odd number `b`s. Therefore, M-INV holds.

24.7.2 Proof the `L = L(EVEN-A-ODD-B)`

The proof that `L(EVEN-A-ODD-B) = L` is divided into two lemmas (i.e., two parts):

1. w∈L ⇔ w∈L(M)
2. w∉L ⇔ w∉L(M)

Lemma 3 *w∈L ⇔ w∈L(M)*

Proof
(⇒) Assume w∈L.

w∈L means that w has an even number of `a`s and an odd number of `b`s. The proof that state invariants hold when w is consumed means that M can only halt in N, which is a final state. Therefore, w∈L(M).

(⇐) Assume w∈L(M).

w∈L(M) means that M halts in N. N's invariant guarantees that w has an even number of `a`s and an odd number of `b`s. Therefore, w∈L. □

Lemma 4 *w∉L ⇔ w∉L(M)*

Proof
(⇒) Assume w∉L.

w∉L means that w does not have an even number of `a`s and an odd number of `b`s. Given that the state invariants always hold, M does not halt in N after consuming w. Since N is the only final state, we have that w∉L(M).

(⇐) Assume w∉L(M).

M does not halt in N (the only final state). Given that the state invariants always hold, this means that w does not have an even number of `a`s and an odd number of `b`s. Therefore, w∉L.

Theorem 4 *L = L(EVEN-A-ODD-B)*

Proof
Lemmas 3 and 4 establish the theorem. □

1 Let Σ `= {a b}`. Design and implement a `dfa` for the following language:

```
L = {w | w has an even number of b}
```

Follow all the steps of the design recipe for state machines.

2 Let Σ = {a b}. Design and implement a `dfa` for the following language:

L = {w | w does not have two consecutive a}

Follow all the steps of the design recipe for state machines.

3 Let Σ = {a b}. Design and implement a `dfa` for the following language:

L = {w | w has an even number of a and an even number of b}

Follow all the steps of the design recipe for state machines.

4 Let M be a `dfa`. When is `'()` in L(M)? Prove your answer.

5 Let Σ = {a b c}. Design and implement a `dfa` for the following language:

L = {w | there is a single Σ element not in w}

Follow all the steps of the design recipe for state machines.

6 Let Σ = {x a e i o u}. Design and implement a `dfa` for the following language:

L = {w | each vowel occurs once and the vowels occur in alphabetical order}

Follow all the steps of the design recipe for state machines.

7 Design and implement a `dfa` for the following language:

L = {w | w represents a proper binary number}

Follow all the steps of the design recipe for state machines. Remember that a proper nonzero binary number does not have leading zeroes.

8 Let Σ = {a b}. Design and implement a `dfa` for the following language:

L = {w | w does not have two consecutive elements that are the same}

Follow all the steps of the design recipe for state machines.

9 Let Σ = {a b}. Design and implement a `dfa` for the following language:

L = {w | w has more than two a}

Follow all the steps of the design recipe for state machines.

10 Design and implement a `dfa` for the following language:

`L = {w | w represents a proper even binary number}`

Follow all the steps of the design recipe for state machines. Remember that a proper nonzero binary number does not have leading zeroes.

11 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L = {w | w`$\in\Sigma^*$`}`

Follow all the steps of the design recipe for state machines.

12 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L = {w | w at least two a and two b}`

Follow all the steps of the design recipe for state machines.

13 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L = {w | w has an odd number of a and ends with a b}`

Follow all the steps of the design recipe for state machines.

14 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L = {w | w's even positions are a}`

For `|w| = n`, the positions are in `[0..n-1]`. For example, `'(a a a b a b)`$\in$`L` and `'(a a b b a)`$\notin$`L`. Follow all the steps of the design recipe for state machines.

15 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L = {w | |w|`$\leq$`4}`

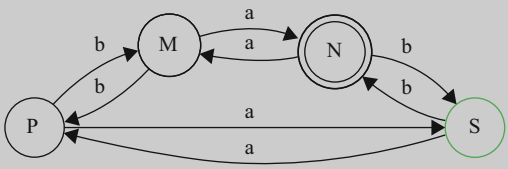
Follow all the steps of the design recipe for state machines.

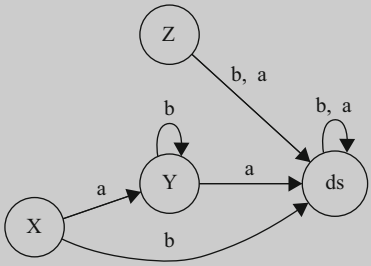
16 Let $\Sigma = \{$`a b`$\}$. Design and implement a `dfa` for the following language:

`L =` $\varnothing$

Follow all the steps of the design recipe for state machines.

17 Mr. Hacker claims to have designed his own version of `EVEN-A-ODD-B`:





His claim is highly suspicious because all he has done is add states that are unreachable from the start state. Design and implement a `dfa` constructor that takes as input a `dfa`, `M`, and that returns the `dfa`, `M'`, obtained by removing the unreachable states from `M`. Prove that `L(M) = L(M')`. Why is it important to remove unreachable states?

25 Applications

There are many practical applications that `dfa`s are well-suited to perform. In the field of programming languages, for example, `dfa` s may be used to recognize tokens. Imagine that in some programming language, an assignment statement is written as follows:

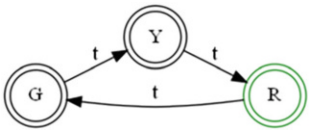
```
index ::= 2 * upper
```

Such an expression must first be parsed to create a parse tree that may then be used by an interpreter or a compiler. Part of parsing is tokenization: the process of identifying the elements (or, if you like, the words) in a program or statement. The above statement would result in five tokens like the following:

`identifier` `assignment` `number` `*` `identifier`

How are can the elements of each token type be determined? A `dfa` may be defined to recognize each token type. The `dfa` that accepts a given program element defines the token type.

Another application `dfa`s are well-suited for the implementation of sequential circuits. For example, a traffic light changes color every time it reads a clock tick repeatedly moving from red to green to yellow and back to red. The following state diagram represents a traffic light:



Every time a clock tick, `t`, is read, the machine changes state. `R`, `Y`, and `G` represent, respectively, the red, the yellow, and the green light is on.

25.1 Finding a Pattern

One of the best known applications of `dfa`s is finding a pattern (or lack of a pattern) in a word. For instance, in Fig. 18, `NO-ABAA` was developed to determine if a word lacks the pattern `'(a b a a)`. Finding a pattern is a common operation that you perform, for example, every time you use `Ctrl-F` to search for a word in a text document. You may ask yourself why should anyone care because designing and implementing `NO-ABAA` required the implementation of the `contains?` function. Clearly, as done in all `NO-ABAA`'s state invariant predicates, to detect that the prohibited pattern is not found, you can simply use:

```
(not (contains? ci PROHIBITED-PATTERN))
```

Why go through all the "trouble" of designing `NO-ABAA` when designing and using `contains?` suffices? This is, indeed, a good question that deserves a scientific answer.

To answer this question, we compare the worst time performance of the two designs. For the comparison to be fair, `NO-ABAA`'s transition function is transformed into a conditional expression used by a function to consume the given word. That is, we eliminate the overhead introduced by using `sm-apply`. The word-consuming function, `consume`, takes as input a state and a word to consume. Each stanza in the conditional expression, other than the first, corresponds to a transition rule in `NO-ABAA`. If the given word is empty, then the given state is examined. If it is a final state, true is returned. Otherwise, false is returned. If the given word is not empty, then a transition rule is simulated. For instance, `'(S a A)` is simulated, when the given state is `'S` and the first word element is `'a`, by recursively calling `consume` with `'A` and

Fig. 22 The function to determine if a word does not contain `'(a b a a)`

```
;; word → Boolean
;; Purpose: Determine if the given word does not contain abaa
(define (contains-no-abaa? w)

  (define FINALS '(S A B C))

  ;; state word → Boolean
  ;; Purpose: Determine if the given word does not contain abaa
  (define (consume s w)
    (cond [(empty? w) (if (member s FINALS) #t #f)]
          [(and (eq? s 'S) (eq? 'a (first w))) (consume 'A (rest w))]
          [(and (eq? s 'S) (eq? 'b (first w))) (consume 'S (rest w))]
          [(and (eq? s 'A) (eq? 'a (first w))) (consume 'A (rest w))]
          [(and (eq? s 'A) (eq? 'b (first w))) (consume 'B (rest w))]
          [(and (eq? s 'B) (eq? 'a (first w))) (consume 'C (rest w))]
          [(and (eq? s 'B) (eq? 'b (first w))) (consume 'S (rest w))]
          [(and (eq? s 'C) (eq? 'a (first w))) (consume 'R (rest w))]
          [(and (eq? s 'C) (eq? 'b (first w))) (consume 'B (rest w))]
          [(and (eq? s 'R) (eq? 'a (first w))) (consume 'R (rest w))]
          [else (consume 'R (rest w))]))
   (consume 'S w))

;; Tests for contains?
(check-equal? (contains-no-abaa? '()) #t)
(check-equal? (contains-no-abaa? '(a b b a a)) #t)
(check-equal? (contains-no-abaa? '(b b b a b a b b a b a)) #t)
(check-equal? (contains-no-abaa? '(a b a a)) #f)
(check-equal? (contains-no-abaa? '(a b b b a b a a a)) #f)
(check-equal? (contains-no-abaa? '(a b a b a a a b a a b)) #f)
```

the rest of the given word. The function to determine if a given word does not contain `'(a b a a)` calls `consume` with `NO-ABAA`'s starting state and the given word. The result of this transformation is displayed in Fig. 22. Observe that the unit tests are written using the same words used to test `contains?` with `PROHIBITED-PATTERN`.

It is now fair to compare the two implementation strategies. To do so, we use words for `contains?`'s worst case and best case. The worst case is when the given word does not contain `'(a b a a)` and `contains?` must traverse the whole word. The best case is when `'(a b a a)` is at the given word's beginning and `contains?` does not need to traverse the given word beyond its first four elements. In comparison, `contains-no-abaa` always traverses the given word in its entirety. The following testing words for `contains?` worst case and best case are defined:

```
(define WORST-CASE (build-list 100000 (λ (i) 'a)))

(define BEST-CASE (append '(a b a a)
                          (build-list 100000 (λ (i) 'a))))
```

To measure execution time, use `time`, which takes as input an expression to evaluate, as follows:

```
(define T1 (time (contains? WORST-CASE PROHIBITED-PATTERN)))
(define T2 (time (contains-no-abaa? WORST-CASE)))

(define T3 (time (contains? BEST-CASE PROHIBITED-PATTERN)))
(define T4 (time (contains-no-abaa? BEST-CASE)))
```

Variables are only defined to prevent the results of the function calls from being printed to the interactions window. The `time` function prints in the interactions window the CPU time, the real time, and the garbage collection time in milliseconds. The results obtained from running the experiments are:[8]

```
cpu time: 7109 real time: 7202 gc time: 46
cpu time: 0    real time: 6    gc time: 0
cpu time: 0    real time: 0    gc time: 0
cpu time: 15   real time: 11   gc time: 0
```

Look at the CPU time. In `contains?`'s worst-case scenario, `contains-no-abaa?` is by far superior: approximately 7 seconds versus 0 seconds. In `contains?`'s best-case scenario, the CPU times are virtually indistinguishable: 15 milliseconds is closer to 0 seconds than to 1 second (or half a second). Clearly, going through all the "trouble" of designing `NO-ABAA` has paid off. This is why you ought to care about designing deterministic finite-state machines.

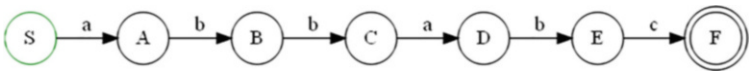
25.2 Generalizing Pattern Detection

As computer scientists, it is clearly unrealistic (and undesirable) to spend our time designing a `dfa` for every search pattern possible given an alphabet. After all, the set of patterns is (countably) infinite. This means that the `dfa` creating process must be automated. If such a function can be written, it needs as input the pattern and the input alphabet, and it returns a `dfa` to determine if a word contains the given pattern.

25.2.1 Design Idea

Given a pattern, `patt`, and an alphabet, `sigma`, the `dfa` built needs |`patt`|`+1` states. These states form the `dfa`'s backbone and lead from the starting to the final state consuming the pattern. For instance, the `dfa` backbone for the pattern `'(a b b a b c)` has the following structure:

[8] Timing data may vary from one execution to the next and from one computer to another.



This, of course, is not the complete `dfa` because it is missing the transitions for when the next symbol in a given word does not match the next symbol in the pattern. For example, if the machine is in state `E` and the next symbol in the input word is not `c`, what state should the machine move to?

When a symbol does not match, the function `contains?` always moves to the next symbol and tries to start matching the pattern from the beginning. For example, assume the input word looks as follows:

```
... a b b a b b a ...
```

The function `contains?` determines that the first six symbols do not match the pattern and repeats the process starting with the next symbol in the input word:

```
... b b a b b a ...
```

Observe that this approach is wasteful. That is, it does not exploit the knowledge accumulated, mainly, that the last three symbols of the six symbols compared in the input word match the first three symbols of the pattern. A `dfa` cannot read the first six symbols and then (wastefully) go back in the tape to try to find a match starting with the first `b` in the input word. Instead, it must decide what state to move to after reading the first six symbols.

Reason about what the states mean (i.e., their invariant properties). For the machine's backbone above, we can state:

`S` Nothing in the pattern has been matched.
`A` a has been matched.
`B` ab has been matched.
`C` abb has been matched.
`D` abba has been matched.
`E` abbab has been matched.
`F` abbabc has been matched.

What state should the machine be in after reading `'(a b b a b b)`? Observe that the longest suffix of the read input that matches the beginning of the pattern is `'(a b b)`. Therefore, the machine needs to move to state `C`.

How are the transition rules computed? We say that the part of the pattern matched for each state represents the core prefix of the state. For each state, it is the word matched in the state's invariant property. For `B`, the core prefix is `'(a b)`, and for `E`, the core prefix is `'(a b b a b)`. We use core prefixes to compute the transitions needed for each state. Assume the states are kept in a list such that the states appear from left to right (i.e., in the direction of the arrows in the backbone). For the backbone above, `states = '(S A B C D E F)`. Let us compute the transitions out of `E`. The core prefix for `E` informs us that `'(a b b a b)` are the last five input symbols read. The next

input symbol may be a, b, or c. For each, we need to identify the longest suffix that matches the pattern's beginning. For a, we have:

```
Last 6 symbols of consumed input: '(a b b a b a)
```

To find the longest matching suffix, compare successively shorter suffixes of the consumed input with the pattern's beginning until a match is found or the suffix is empty. This search is outlined as follows:

```
Pattern: '(a b b a b c)
 Suffix: '(a b b a b a) → does not match
         '(b b a b a)   → does not match
         '(b a b a)     → does not match
         '(a b a)       → does not match
         '(b a)         → does not match
         '(a)           → match
```

The longest matching suffix with the pattern's beginning is a. This is A's core prefix. This means that the machine must transition to A. The needed transition is (E a A). Let us repeat this process for b:

```
Pattern: '(a b b a b c)
 Suffix: '(a b b a b b) → does not match
         '(b b a b b)   → does not match
         '(b a b b)     → does not match
         '(a b b)       → match
```

The longest matching suffix with the pattern's beginning is C's core prefix. The machine needs to transition to C, and the needed transition is (E b C). Finally, let us repeat the process for c:

```
Pattern: '(a b b a b c)
 Suffix: '(a b b a b c) → match
```

The longest matching suffix with the pattern's beginning is F's core prefix. Therefore, the needed transition is (E c F).

Have you noticed the pattern for the destination state in each of the computed transition rules? It is always (list-ref states (length lsuffix)), where lsuffix is the longest matching suffix. Using our example, it is illustrated as follows:

```
(list-ref states (length '(a))) = A
(list-ref states (length '(a b b))) = C
(list-ref states (length '(a b b a b c))) = F
```

Based on our design idea, we proceed to implement the needed functions.

25.2.2 The `contains-pattern?` Predicate

The predicate to determine if a pattern appears in a given word takes as input two words and an input alphabet. We assume that the given words are strictly composed of an arbitrary number of elements from the given alphabet. It builds the `dfa` to detect the given pattern is present in the given word using the given alphabet. It then applies the built `dfa` to the given word to return the appropriate Boolean. Following the steps of the design recipe for functions yields:

```
;; word word alphabet → Boolean
;; Purpose: Determine if the given first word is in
;;          the given second word
;; Assume: Given words in sigma*
(define (contains-pattern? patt text sigma)
  (let [(M (build-pattern-dfa patt sigma))]
    (eq? (sm-apply M text) 'accept)))

;; Tests for contains-pattern?
(check-equal? (contains-pattern? '(a b b a b c) '() '(a b c))
              #f)
(check-equal?
 (contains-pattern?
   '(a b b a b c) '(a b c a a a b b a b b c b a a b) '(a b c))
 #f)
(check-equal?
 (contains-pattern? '(a b b a b c) '(a b b a b c) '(a b c))
 #t)
(check-equal?
 (contains-pattern? '(a b b a b c)
                    '(a a b b a b a b b a b c a c c c c)
                    '(a b c))
 #t)

(check-equal? (contains-pattern? '(b a a) '() '(a b)) #f)
(check-equal? (contains-pattern? '(b a a)
                                 '(a b a b a b b a b b c b a b)
                                 '(a b))
              #f)
(check-equal? (contains-pattern? '(b a a)
                                 '(b a a)
                                 '(a b))
              #t)
```

```
(check-equal? (contains-pattern? '(b a a)
                                 '(a a b b a b a b b a a b a)
                                 '(a b))
              #t)
```

25.2.3 The `build-pattern-dfa` Constructor

For a given pattern, `patt`, and a given input alphabet, `sigma`, the goal is to build a `dfa` for the following language:

```
L = {w | w contains patt}
```

The constructor needs to:

1. Generate the states for the new `dfa`
2. Compute the core prefix for each state
3. Compute the transitions for the new `dfa`

Once these are computed, a new `dfa` is constructed and returned.

We choose to define the first state in the list of generated states as the starting state and the last state generated as the final state. To generate a state, the `FSM` function `generate-symbol` is used. It takes as input a seed symbol that the generated symbol shall start with and a nonempty list of symbols (including the seed) that the generated symbol may not equal.

The generation of the core prefixes may be done so as to correspond with the list of generated states. That is, the first prefix is for the first state, the second prefix is for the second state, and so on. This requires building a list of length one greater than the length of the pattern. For each natural number, `i`, in `[0..(sub1 (length patt))]`, the core prefix for the $\texttt{i}^{\texttt{ith}}$ state is given by taking the first `i` elements of the pattern.

To generate the transition function, the needed transitions for each state, `s`, may be generated using the states, the input alphabet, the core prefix for `s`, and the pattern. All the generated rules are appended to define the transition function for the new `dfa`.

Following the steps of the design recipe for functions produces the constructor displayed in Fig. 23. To generate the states and the core prefixes, `build-list` is used to build a list of the right length according to the design idea above. The generation of the transitions for a state and its core prefix is left to an auxiliary function. The new `dfa` is constructed using the generated states, the given alphabet, the first generated state as the starting state, the last generated state as the only final state, and the transitions generated by the auxiliary function. Given that a transition function is computed, `'no-dead` is given as an argument to `make-dfa` to suppress the addition of a dead state. Finally, to test the new constructor, two `dfa`s are constructed and used in the tests in conjunction with `sm-apply`.

Fig. 23 The `dfa` constructor to detect a pattern in a given word

```
;; word alphabet → dfa
;; Purpose: Build a dfa for L = all words that contain the
;;                              given pattern
(define (build-pattern-dfa patt sigma)
  (let* [(sts (build-list (add1 (length patt))
                          (λ (n) (generate-symbol 'A '(A)))))
         (core-prefixes (build-list (add1 (length patt))
                                    (λ (i) (take patt i))))
         (deltas (append-map
                  (λ (s cp)
                    (gen-state-trans s sts sigma cp patt))
                  sts
                  core-prefixes))]
    (make-dfa sts
              sigma
              (first sts)
              (list (last sts))
              deltas
              'no-dead)))

;; Tests for build-pattern-dfa
(define M (build-pattern-dfa '(a b b a) '(a b)))
(define N (build-pattern-dfa '(a d) '(a b c d)))

(check-equal? (sm-apply M '()) 'reject)
(check-equal? (sm-apply M '(a a b b b a)) 'reject)
(check-equal? (sm-apply M '(b b b a a a b b)) 'reject)
(check-equal? (sm-apply M '(a b b a)) 'accept)
(check-equal? (sm-apply M '(b b a a a b b a b b a)) 'accept)
(check-equal? (sm-apply M '(a b b b a b b a)) 'accept)

(check-equal? (sm-apply N '()) 'reject)
(check-equal? (sm-apply N '(a b c d a b c c)) 'reject)
(check-equal? (sm-apply N '(c c b a b d)) 'reject)
(check-equal? (sm-apply N '(a d)) 'accept)
(check-equal? (sm-apply N '(b c a a d c c b)) 'accept)
(check-equal? (sm-apply N '(c d b c a d c a d)) 'accept)
```

25.2.4 The `gen-state-trans` Function

The function to generate the transitions for a state requires a state, the list of states, the input alphabet, the given state's core prefix, and the pattern for the new `dfa` to match. For each element of the given input alphabet, a transition is generated using the given state, a word that starts with the given core prefix and ends with the alphabet element (i.e., the last elements read), the pattern, the states (any of which may be transitioned to), and the last element read from the tape (i.e., the alphabet element). The generation of the rule is left to an auxiliary function that traverses the last elements read to find the longest suffix that matches the beginning of the pattern. The

alphabet element (i.e., the last element) must be provided to the auxiliary function because when nothing is matched, the needed transition generated takes the machine to the starting state consuming the given alphabet element.

Following the steps of the design recipe for functions leads to:

```
;; state (listof state) alphabet word word → (listof dfa-rule)
;; Purpose: Generate failed match transitions for the
;;          given state
(define (gen-state-trans s states sigma cp patt)
  (map (λ (a)
         (gen-state-tran s (append cp (list a)) patt states a))
       sigma))

;; Tests for gen-state-trans
(check-equal?
  (gen-state-trans 'E
                   '(S A B C D E F)
                   '(a b c)
                   '(a b b a b)
                   '(a b b a b c))
  '((E a A) (E b C) (E c F)))
(check-equal?
  (gen-state-trans 'S
                   '(S A B C D E F)
                   '(a b c)
                   '()
                   '(a b b a b c))
  '((S a A) (S b S) (S c S)))

(check-equal?
  (gen-state-trans 'S '(S A F) '(a b) '() '(a b))
  '((S a A) (S b S)))
(check-equal?
  (gen-state-trans 'A '(S A F) '(a b) '(a) '(a b))
  '((A a A) (A b F)))
(check-equal?
  (gen-state-trans 'F '(S A F) '(a b) '(a b) '(a b))
  '((F a F) (F b F)))
```

Observe that the tests display all the transitions out of the given state that are needed. There is one rule for each element of the input alphabet. Doing this for each state guarantees that the result is a transition function.

The function to generate a `dfa`-rule for a given state and a given word to match traverses the word to match. If the given word is empty, then a transition from the given state to the starting state that consumes the given last element read is generated. If the length of what is to be matched is longer

than the pattern, then a match with a prefix of the pattern is not possible. Remember that the word to match is formed by adding a symbol to the end of the state's core prefix. This means that the new `dfa`, when executed, will be in its final state and, therefore, the transition must be to the final state (i.e., the given state or equivalently the last state in the given list of states). If the word to match is equal to the front of the pattern, then, as per the design idea, the needed transition is to the state indexed by the length of the word to match. If these three conditions fail, then the rest of the word to match is recursively processed to determine the needed transition. The following function is obtained by following the steps of the design recipe for functions:

```
;; word word word (listof state) symbol → dfa-rule
;; Purpose: Generate dfa rule for given state and given word
;;          to match in the given pattern
(define (gen-state-tran s to-match patt states last-read)
  (cond [(empty? to-match) (list s last-read (first states))]
        [(> (length to-match) (length patt))
         (list s last-read s)]
        [(equal? to-match (take patt (length to-match)))
         (list s
               (last to-match)
               (list-ref states (length to-match)))]
        [else (gen-state-tran s
                              (rest to-match)
                              patt
                              states
                              last-read)]))

;; Tests for gen-state-tran
(check-equal?
  (gen-state-tran
    'C '(a b b b) '(a b b a b c) '(S A B C D E F) 'b)
  '(C b S))
(check-equal?
  (gen-state-tran
    'S '(b) '(a b b a b c) '(S A B C D E F) 'b)
  '(S b S))
(check-equal?
  (gen-state-tran
    'S '(a) '(a b b a b c) '(S A B C D E F) 'a)
  '(S a A))
(check-equal?
  (gen-state-tran
    'D '(a b b a c) '(a b b a b c) '(S A B C D E F) 'c)
  '(D c S))
```

```
(check-equal?
  (gen-state-tran
    'E '(a b b a b b) '(a b b a b c) '(S A B C D E F) 'b)
  '(E b C))
```

This completes the design of a predicate to determine if a pattern occurs in a word. Run the program and make sure all the tests pass. This algorithm is the basis for the efficient and widely implemented Knuth-Morris-Pratt (`KMP`) algorithm. The `KMP` algorithm is a string matching algorithm that looks for occurrences of a string (i.e., a pattern) in a block of text (i.e., a word). Given that most programming languages do not have a `dfa` type like `FSM`, the KMP algorithm represents the `dfa` differently. It uses a vector of indices into the pattern to represent where matching ought to continue when the next element in the text does not match the next element in the pattern (i.e., equivalent to the transitions that move to a state closer to the starting state in the `dfa` built in our implementation). You are strongly encouraged to review the `KMP` algorithm.

18 Define and implement a `dfa` for a traffic light.

19 Design and implement a predicate to determine if a given pattern does not occur in a given word for a given alphabet.

20 In computational biology, DNA is a chain of complementary nucleotide bases. The bases are adenine (`A`), cytosine (`C`), guanine (`G`), and thymine (`T`). Under normal circumstances, `A` pairs with `T`, and `C` pairs with `G`. We say that a DNA chain is normal if all pairs are formed as outlined. For example, consider the following DNA strands:

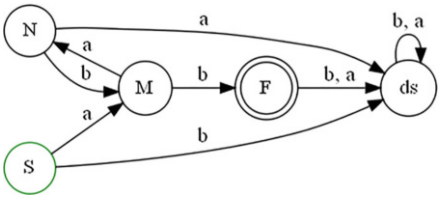
```
AT AT GC CG TA
CG AT TA TG AT GC
```

The first is normal because `A` and `T` only pair with each other and `C` and `G` only pair with each other. The second is not normal because it contains `TG`.

Design and implement a `dfa`-based program to determine if a given DNA strand is normal. *Hint*: You need to represent all possible pairing of bases as elements in an input alphabet. For example, `AT` and `TA` may be represented as `a`, and `TG` and `GT` may be represented as `z`. With such a mapping, any input DNA strand may be transformed into a word using your alphabet symbols.

Chapter 6
Nondeterministic Finite-State Machines

We have seen that a `dfa` can decide a language (i.e., determine if a word is or is not in a language) whose words are built using concatenation and Kleene star. Consider, for example, the following transition diagram for a `dfa`:



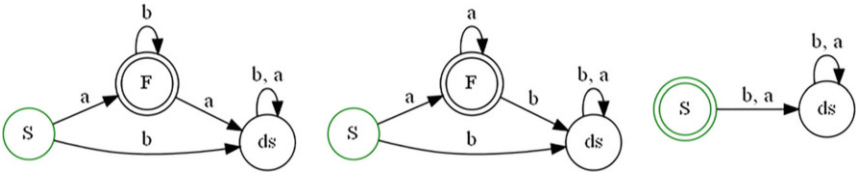
If you think about it, a transition in a `dfa` represents concatenating an alphabet symbol, and a loop is concatenating a value generated by a Kleene star – the loop may be entered 0 or more times. The language of the machine above is $\texttt{a(ab)}^*\texttt{b}$. This is not a proof, but intuitively it appears that if it is generated by concatenation or Kleene star, then it can be decided by a `dfa`.

What about deciding a regular language that requires union? When there is a union, words are generated by randomly selecting a branch in the union of regular expressions. For instance, consider the following language:

$$\texttt{L} = \texttt{ab}^* \cup \texttt{aa}^* \cup \epsilon$$

There are three types of words that may be generated. It is not difficult to build a `dfa` for the language represented by each regular expression choice in the union:

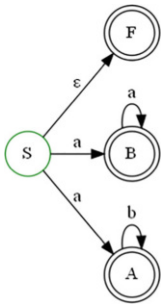
M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_6



It is difficult, however, to see how `L` can be decided by a `dfa`. When started, the `dfa` would have to arbitrarily choose which type of word to decide without reading any of the input. This is impossible for a `dfa` to do because it has a transition function. A `dfa` always carries out the same computation given the same input. It cannot arbitrarily choose how to carry out a computation.

26 Nondeterministic Finite-State Machines

A new model of a computer is needed, one that allows a machine to change state in a manner that is not fully determined by the transition relation. When the machine has a choice, it nondeterministically chooses which transition (or transitions as we shall see) to use. For instance, a finite-state machine for `L` may look as follows:



When the machine above starts, without reading anything from the tape, it nondeterministically decides how to process the input. It processes it as a word that may or may not be either in $\{\epsilon\}$, in `aa`*, or in `ab`*. Observe that there are two new characteristics:

- A nondeterministic machine may change state without consuming anything from the input using one or more ϵ-transitions such as `(S ϵ F)`.
- From a given state, there may be more than one transition on a given alphabet element like `(S a B)` and `(S a A)`.

This means that the transition relation is no longer a function. That is, given the same input, the machine may potentially carry out one of several different computations. Processing `'(a b b)`, for example, can be potentially be done in three different ways:

```
((a b b) S) ⊢ ((a b b) F)

((a b b) S) ⊢ ((b b) B)

((a b b) S) ⊢ ((b b) A) ⊢ ((b) A) ⊢ (() A)
```

The first two computations would reject `'(a b b)` because the machine does not halt in a final state with the input empty. The third computation accepts `'(a b b)` because the machine halts in a final state with the input empty. Based on this, is `'(a b b)` in `L` or not? We say that a word is in the language of a nondeterministic finite-state machine if there is at least one potential computation that leads to accept. If there are no potential computations that lead to accept, then the input word is not in the language of the machine. This means that `'(a b b)` is in the language of the machine. You may ask yourself how can this possibly work. How can the machine determine which computation to carry out? You may assume that if the input word is in the machine's language, then the machine can sense which is the computation that leads to accept. For every nondeterministic choice made during such a computation, the machine can sense which is the correct transition to use. Machines with such "intuition" sound very powerful, no?

Formally, a *nondeterministic finite-state automaton*, `ndfa`, is defined as follows:

```
A nondeterministic finite-state automaton, ndfa, is a

    (make-ndfa K Σ S F δ)
```

That is, an `ndfa` is a type in `FSM`. The inputs to this constructor are the same as for `make-dfa` except for δ. Here, δ is a transition relation, not a function, that may have ϵ-transitions and multiple transitions from a state on the same alphabet element. An `ndfa` (transition) rule is defined as follows:

```
(Q a R), where Q,R∈K ∧ a∈{Σ ∪ {ϵ}}
```

Given a configuration `(m r)`, where `m`$\in \Sigma^*$ and `r`$\in$`K`, the machine may have no transitions it may follow and halts or may move to one of several different configurations because of ϵ-transitions or multiple transitions from `r` on the same alphabet element. A word, `w`, is accepted by an `ndfa`, `N`, if there exists a computation such that:

```
(w s) ⊢* (() f), where f∈F
```

It does not matter if there are transitions, consuming all or part of `w`, that do not lead to accept or if there are multiple computations that lead to accept.

Fig. 24 An `ndfa` for $\mathtt{L} = \{\epsilon\} \cup \mathtt{aa}^* \cup \mathtt{ab}^*$

```
#lang fsm

;; L = {ε} ∪ aa* ∪ ab*
(define LNDFA (make-ndfa '(S A B F)
                         '(a b)
                         'S
                         '(A B F)
                         `((S a A)
                           (S a B)
                           (S ,EMP F)
                           (A b A)
                           (B a B))))

;; Tests for LNDFA
(check-equal? (sm-apply LNDFA '(a b a)) 'reject)
(check-equal? (sm-apply LNDFA '(b b b b b)) 'reject)
(check-equal? (sm-apply LNDFA '(a b b b b a a a)) 'reject)
(check-equal? (sm-apply LNDFA '()) 'accept)
(check-equal? (sm-apply LNDFA '(a)) 'accept)
(check-equal? (sm-apply LNDFA '(a a a a)) 'accept)
(check-equal? (sm-apply LNDFA '(a b b)) 'accept)
```

A word is in the language of `N` if there is at least one computation that leads to accept. Finally, the language of `N`, `L(N)`, is all the words accepted by `N`.

A natural question to ask is how does an `ndfa` choose which nondeterministic transition to make. For instance, consider the following transition rules:

```
(P a Q)      (P a R)      (R EMP T)
```

When in state `P` and reading an `a`, the `ndfa` has a nondeterministic choice to make. Does it move to `Q` or to `R`? When in `R`, does the machine stay in `R` or does it move to `T` without consuming any input? The answers to these questions are the same. When faced with a nondeterministic choice, an `ndfa` only makes transitions that lead to accept. If no such transition is possible, then the machine halts and decides to accept or reject as outlined above. We shall not concern ourselves with how a nondeterministic machine knows if making a nondeterministic transition will lead to accept. This is the power of nondeterministic machines. All we need to understand is that a nondeterministic machine only makes nondeterministic transitions that lead to accept. If none of the potential computations lead to accept, the machine senses it and rejects.

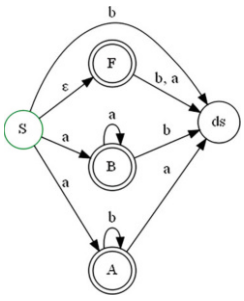
An immediate observation that can be made is that a `dfa` is an `ndfa`. It is an `ndfa` that has no ϵ-transitions and that has exactly one transition out of a state for each element of the alphabet. That is, a `dfa` is a finite-state automaton that does not use nondeterminism. Clearly, not every `ndfa` is a `dfa`.

To illustrate the use of the constructor, Fig. 24 displays the implementation of an `ndfa` for `L`. Observe that the only transitions listed are those that are

Fig. 25 An `ndfa` with explicit dead-state transitions for L

```
;; L = ϵ U aa* U ab*
(define LNDFA (make-ndfa `(S A B F ,DEAD)
                         '(a b)
                         'S
                         '(A B F)
                         `((S a A)
                           (S a B)
                           (S ,EMP F)
                           (A b A)
                           (B a B)
                           (S b ,DEAD)
                           (A a ,DEAD)
                           (B b ,DEAD)
                           (F a ,DEAD)
                           (F b ,DEAD))))
```

(a) Implementation.



(b) Transition diagram.

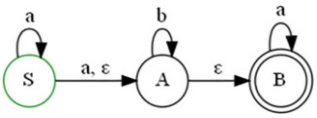
on a path to an accepting state. The tests are written in the same manner as done for a `dfa` using `sm-apply`. As with a `dfa`, random testing is done using `sm-test`:

```
> (sm-test LNDFA 10)
'(((b b b a a a a a) reject)
  ((a a b b b a) reject)
  ((b b a a a a a a b) reject)
  ((a a b) reject)
  ((a a b a b a b b a) reject)
  ((a a a a) accept)
  ((a b) accept)
  ((b a a a a) reject)
  (() accept)
  ((a b a a b a a) reject))
```

It is not uncommon at the beginning to feel uncomfortable with missing transitions. You may ask yourself, what does the `LNDFA` do if it is in state `S` and the next input symbol is `b`? There are two equivalent ways to think about this. The first is, as mentioned earlier, the machine halts and, in this case, rejects because all the input is not consumed. The second is to understand that the transitions to the dead state are implicit. Under such a view, the machine moves to the dead state, consumes the rest of the input, and rejects (because the dead state is not a final state). You are, of course, always free not to use the shorthand notation and always explicitly include transitions to the dead state as displayed in Fig. 25. Most programmers eventually prefer the shorthand notation for creating an `ndfa`.

27 Designing an `ndfa`

Designing an `ndfa` can prove easier than designing a `dfa`, but care must be taken when reasoning about the machine. When an input symbol is processed, the machine may end in more than one state. Consider, for example, the following `ndfa`:



What state does the machine move to if it is in `S` and consumes an `a`? Due to nondeterminism, there is no way we can actually know. We do know, however, the possible states the machine can be in are not completely random. After consuming an `a`, the machine can be in `S`, in `A`, or in any state reachable by only following ϵ-transitions out of `S` or `A`. Specifically, after consuming the `a`, the machine may end in `S`, `A`, or `B`. As a designer of `ndfa`s, this is important to understand because the design must guarantee that the state invariant holds for any state in a computation that leads to accept after a transition that requires reading from the tape is followed.

To aid us in reasoning about `ndfa`s, we define the *empties* of a state `R`, `E(R)`, as follows:

```
E(R) = {{R} ∪ {P | ((ϵ R) ⊢* (ϵ P))}}
```

That is, `E(R)` contains `R` and all states reachable from `R` by only following ϵ-transitions. Returning to the sample `ndfa` above, we have that:

```
E(S) = {S A B}
E(A) = {A B}
E(B) = {B}
```

This means that when a transition is made into a state, `R`, after reading an input element, we must prove the state invariants hold for every state in `E(R)` in a computation that leads to accept. For instance, for `(S a S)`, we must prove state invariants for `S` if only `(S a S)` is used, for `S` and `A` if only `(S a S)` and `(S EMP A)` are used, and for `B` if only `(S a S)`, `(S EMP A)`, and `(A EMP B)` are used.

To illustrate designing an `ndfa`, assume that the input alphabet is {`a b c`} and that the language that the machine decides is:

```
L = {w | a∉w ∨ b∉w ∨ c∉w}
```

The following subsections outline the results obtained for each step of the design recipe for state machines.

27.1 Name, Alphabet, and Tests

The machine must accept all words and only the words that have at least one alphabet member missing. A descriptive name for the `ndfa` is `AT-LEAST-ONE-MISSING`. The input alphabet is {`a b c`}.

The following tests illustrate the expected behavior:

```
;; Tests for AT-LEAST-ONE-MISSING
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a b c)) 'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b b a b c b a))
              'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b a c)) 'reject)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '()) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(c)) 'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(c c a a))
              'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(b b c b b b))
              'accept)
(check-equal? (sm-apply AT-LEAST-ONE-MISSING '(a a a b b b))
              'accept)
```

Observe that words that contain all three symbols in the input alphabet are rejected and those that do not are accepted.

27.2 Design Idea and Conditions

The machine nondeterministically decides to process the word as if `a` is missing, as if `b` is missing, or as if `c` is missing. As the word is processed, the consumed input, `ci`, must satisfy one of the four conditions: nothing is consumed, or for each `x`∈`(sm-sigma AT-LEAST-ONE-MISSING)`, `x`∉`ci`. This means four states are needed and these are documented in the program as follows:

```
;; States
;; S: the consumed input is empty, starting state
;; A: the consumed input does not contain a, final state
;; B: the consumed input does not contain b, final state
;; C: the consumed input does not contain c, final state
```

27.3 Transition Relation

From the starting state, `S`, the machine nondeterministically moves to either `A`, `B`, or `C`. In `A`, it processes an arbitrary number of `b`s or `c`s. In `B`, it processes an arbitrary number of `a`s or `c`s. In `C`, it processes an arbitrary number of `a`s or `b`s. The transition relation is:

```
`((S ,EMP A)
  (S ,EMP B)
  (S ,EMP C)
  (A b A)
  (A c A)
  (B a B)
  (B c B)
  (C a C)
  (C b C))
```

27.4 Implementation and Testing

Using the results for the previous steps of the design recipe for state machines, the machine is implemented as follows:

```
(define AT-LEAST-ONE-MISSING (make-ndfa '(S A B C)
                                        '(a b c)
                                        'S
                                        '(A B C)
                                        `((S ,EMP A)
                                          (S ,EMP B)
                                          (S ,EMP C)
                                          (A b A)
                                          (A c A)
                                          (B a B)
                                          (B c B)
                                          (C a C)
                                          (C b C))))
```

Run the machine, and make sure that all the unit tests pass. In addition, use `sm-test` to validate that the machine works correctly.

27.5 State Invariant Predicates

The invariant state predicate for S must determine if the consumed input is empty. Following the steps of the design recipe for functions yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if the given word is empty
(define (S-INV ci) (empty? ci))

;; Test for S-INV
(check-equal? (S-INV '()) #t)
(check-equal? (S-INV '(a b)) #f)
```

The invariant state predicate for A must determine if the consumed input does not contain an a. Following the steps of the design recipe for functions yields the following predicate:

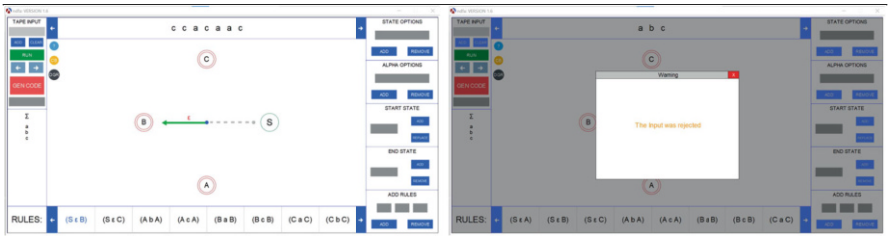
```
;; word → Boolean
;; Purpose: Determine if the given word does not contain a
(define (A-INV ci) (empty? (filter (λ (a) (eq? a 'a)) ci)))

;; Test for A-INV
(check-equal? (A-INV '(a)) #f)
(check-equal? (A-INV '(a c b)) #f)
(check-equal? (A-INV '(c c b a b)) #f)
(check-equal? (A-INV '(b)) #t)
(check-equal? (A-INV '(c c b c b)) #t)
(check-equal? (A-INV '()) #t)
```

The invariant state predicate for B must determine if the consumed input does not contain a b. Following the steps of the design recipe for functions yields the following predicate:

```
;; word → Boolean
;; Purpose: Determine if the given word does not contain b
(define (B-INV ci) (empty? (filter (λ (a) (eq? a 'b)) ci)))

;; Test for B-INV
(check-equal? (B-INV '(b)) #f)
(check-equal? (B-INV '(a c b)) #f)
(check-equal? (B-INV '(a a b a b)) #f)
(check-equal? (B-INV '(c)) #t)
(check-equal? (B-INV '(c c a c c a a a)) #t)
(check-equal? (B-INV '()) #t)
```

The invariant state predicate for C must determine if the consumed input does not contain a c. Following the steps of the design recipe for functions yields the following predicate:

Fig. 26 Visualizations for `AT-LEAST-ONE-MISSING`



(a) Input word in language. **(b)** Input word not in language.

```
;; word → Boolean
;; Purpose: Determine if the given word does not contain c
(define (C-INV ci) (empty? (filter (λ (a) (eq? a 'c)) ci)))

;; Test for C-INV
(check-equal? (C-INV '(c)) #f)
(check-equal? (C-INV '(a b c b)) #f)
(check-equal? (C-INV '(c c b a b)) #f)
(check-equal? (C-INV '(b)) #t)
(check-equal? (C-INV '(b b a a b a a a)) #t)
(check-equal? (C-INV '()) #t)
```

Validate `AT-LEAST-ONE-MISSING`'s design by running it in the `FSM` visualization tool with the invariant state predicates:

```
(sm-visualize AT-LEAST-ONE-MISSING
              (list 'S S-INV)
              (list 'A A-INV)
              (list 'B B-INV)
              (list 'C C-INV))
```

For input words in `L(AT-LEAST-ONE-MISSING)`, the visualization tool traces one, of possibly many, paths that lead to accept. Observe how the initial transition from `S` is always to a correct state as illustrated in Fig. 26a. This is expected because an `ndfa` always makes a move to a state that leads to accept when the input word is in the machine's language. The arrow also changes color to indicate whether that state invariant holds. For input words not in `L(AT-LEAST-ONE-MISSING)`, the visualization tool opens a pop-up window informing you that the input is rejected as illustrated in Fig. 26b. This is because none of the possible computations lead to accept. To visualize all such paths to validate that the input word is not in the machine's language gets unwieldy quickly and is impractical to visualize. It is worth noting that this is also the reason why `show-transitions` only returns `'reject` when given a word that is not in the language of the given nondeterministic machine.

27.6 Correctness

27.6.1 Proving State Invariants Hold

Recall how to prove that a state invariant holds after a dfa transition using the following rule:

```
(R a Q)
```

It is assumed that the consumed input satisfies R's invariant, and it is shown that adding a to the consumed input satisfies Q's invariant. By consuming a, the consumed input becomes longer. For an ndfa, we follow the same basic approach, but we need to be aware that a may be ϵ and, therefore, if such is the case, the consumed input does not become longer. This means that we must show that R's invariant implies the invariant for every state in E(R) that can lead to accept.

Why are we only concerned with nondeterministic transitions that may lead to an accept? Recall that an ndfa never makes a nondeterministic transition unless it furthers a computation that leads to an accept. Therefore, the nondeterministic choices not made do not concern us. This means that to establish correctness, we only need to reason about nondeterministic transitions that can lead to accept. Nondeterministic transitions that cannot lead to an accept are never made and, thus, not part of any computation.

We shall simplify how the proof is written by suppressing the "boilerplate" parts of the proof by induction on the number of transitions made to consume the input word. We shall use this abbreviated strategy to write the proof for all state invariants holding during a computation that leads to an accept performed by AT-LEAST-ONE-MISSING. It is assumed that readers of this textbook now understand that, in essence, the proof argues that the starting state's invariant must hold when no input has been consumed and that the state invariant for a state transitioned into must hold assuming the previous state's invariant holds.

Assume:

- M=AT-LEAST-ONE-MISSING
- w∈(sm-sigma M)*
- F=(sm-finals M)
- ci is the consumed input

Theorem 1 *The state invariants hold when M is applied to w.*

For the base case, we must show that S-INV holds before making any transitions. For the inductive step, we must show that the invariant for the state transitioned into holds.

Proof

When M starts, `S-INV` holds because ci = `'()`. This establishes the base case.

Proof invariants hold after each transition:

`(S ϵ A)`: By inductive hypothesis, S-INV holds. This means ci = `'()`. Using this transition adds nothing to ci and, therefore, ci = ϵ after using this rule. `A-INV` holds because ci contains zero `a`s.

`(S ϵ B)`: By inductive hypothesis, S-INV holds. This means ci = `'()`. Using this transition adds nothing to ci and, therefore, ci = ϵ after using this rule. `B-INV` holds because ci contains zero `b`s.

`(S ϵ C)`: By inductive hypothesis, S-INV holds. This means ci = `'()`. Using this transition adds nothing to ci and, therefore, ci = ϵ after using this rule. `C-INV` holds because ci contains zero `c`s.

`(A b A)`: By inductive hypothesis, A-INV holds. A-INV guarantees that the consumed input does not contain an `a`. Consuming a `b` means that the consumed input remains without an `a`. Therefore, A-INV holds after the transition.

`(A c A)`: By inductive hypothesis, A-INV holds. A-INV guarantees that the consumed input does not contain an `a`. Consuming a `c` means that the consumed input remains without an `a`. Therefore, A-INV holds after the transition.

`(B a B)`: By inductive hypothesis, B-INV holds. B-INV guarantees that the consumed input does not contain a `b`. Consuming an `a` means that the consumed input remains without a `b`. Therefore, B-INV holds after the transition.

`(B c B)`: By inductive hypothesis, B-INV holds. B-INV guarantees that the consumed input does not contain a `b`. Consuming a `c` means that the consumed input remains without a `b`. Therefore, B-INV holds after the transition.

`(C a C)`: By inductive hypothesis, C-INV holds. C-INV guarantees that the consumed input does not contain a `c`. Consuming an `a` means that the consumed input remains without a `c`. Therefore, C-INV holds after the transition.

`(C b C)`: By inductive hypothesis, C-INV holds. C-INV guarantees that the consumed input does not contain a `c`. Consuming a `b` means that the consumed input remains without a `c`. Therefore, C-INV holds after the transition.

This establishes the inductive step. □

27.6.2 L = L(AT-LEAST-ONE-MISSING)

Lemma 1 *w∈L ⇔ w∈L(M)*

Proof
(⇒) Assume w∈L.

w∈L means that w does not contain, at least, an a, a b, or a c. Given that the state invariants always hold, this means M consumes all its input in either A, B, or C. Since A,B,C∈F, w∈L(M).

(⇐) Assume w∈L(M).

w∈L(M) means that M consumes all its input and halts in A, B, or C. Given that the state invariants always hold, we may conclude that w is missing a, b, or c. Therefore, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(M)*

Proof
(⇒) Assume w∉L.

w∉L means that w has at least one a, one b, and one c. Given that state invariants always hold, M cannot halt in A, B, or C after consuming w. Since it cannot halt in a final state after consuming w, w∉L(M).

(⇐) Assume w∉L(M).

w∉L(M) means that M does not halt in a final state consuming all of w. Given that the state invariants always hold, w must have at least one a, one b, and one c. Thus, w∉L. □

Theorem 2 *L = L(AT-LEAST-ONE-MISSING).*

Proof
Lemmas 1 and 4 establish the theorem. □

1 Design and implement a dfa for L(AT-LEAST-ONE-MISSING). Follow all the steps of the design recipe for state machines.

2 Design and implement an ndfa for:

$(ab)^*b^* \cup ab^*$

Follow all the steps of the design recipe for state machines.

3 Design and implement an `ndfa` for:

`L = {w | w∈aa*  ∨ w∈ab*}`

Follow all the steps of the design recipe for state machines.

4 Let Σ ={`a b c`}. Design and implement an `ndfa` for:

`L = {w | w is missing exactly 1 of the elements in` Σ`}`

Follow all the steps of the design recipe for state machines.

5 Let Σ ={`a b`}. Design and implement an `ndfa` with only three states for:

`L = {w | w ends with bb}`

Follow all the steps of the design recipe for state machines.

28 Equivalence of `dfa` and `ndfa`

Designing `ndfa`s can be much simpler than designing a `dfa`. This makes `ndfa`s a valuable design tool for a problem-solver. However, it comes at a cost. How can an `ndfa` be implemented as a function that does not build a machine (as done in Sect. 25.1 for `NO-ABAA`)? It is difficult to see how to translate an `ndfa` in `FSM` into a function in a programming language that does not have an `ndfa` type. The difficulty arises in implementing nondeterminism. How is the right transition always made? A deterministic program cannot make a nondeterministic choice and guarantee that given the same input, the same computation is carried out. A deterministic program would have to simulate all possible computations and determine if any leads to accept. Such a simulation may be implemented by performing a breadth-first search of all possible computations. If a configuration is found that is in a final state with empty input, then accept. If there are no configurations left to explore, then reject. The key to understanding if such a design works is to realize that a configuration is never explored more than once. Therefore, if an accept configuration is not found, then eventually the configurations to explore become empty.

The fact that a deterministic function may be written to simulate an `ndfa` raises an intriguing question: Does endowing a finite-state machine with nondeterminism give us more computational power? That is, is there anything

an ndfa can do that a dfa cannot do? To simulate an ndfa a dfa needs to simulate all computations of the ndfa simultaneously. At first glance, this may sound like preposterous. After all, a dfa cannot be in multiple states at the same time. This is certainly true, but do we need a dfa to be in multiple states at the same time? Consider an ndfa, N = (make-ndfa S Σ s F δ), making a transition out of state P on a$\in \Sigma$. There can be several transition rules out of P on a:

(P a R_1)
⋮
(P a R_n)

After consuming a, what states can N be in? Consider the rule that moves to N to R_i. N can be in any state in E(R_i). That is, N may be in R_i or any state reachable from R_i using only ϵ-transitions. This means that N may be in any of the following states:

E(R_1) $\cup$ E(R_2) $\cup$ ... $\cup$ E(R_n)

Observe that the above is an element in 2^S. Let us call it Z. Think of Z as a *super state* for a dfa that represents all the states N may be in. To simulate all possible computations that may be performed by N, a dfa transitions between super states. After consuming all the input, it accepts if it is in a super state that contains a final state in N. Otherwise, it rejects.

It is necessary to show how such a dfa, M, is constructed and to show that L(M) = L(N). Showing how to build something (like a dfa) and showing that the construction is correct (like L(M) = L(N)) is called a *constructive proof*. A constructive proof has a construction algorithm and a proof of its correctness.

28.1 Building a dfa from an ndfa

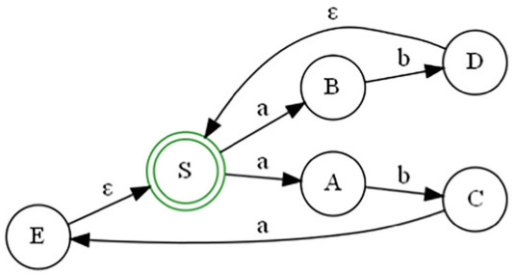
Let N = (make-ndfa S Σ s F δ). Building a dfa from N hinges on computing a transition function between super states for a dfa and encoding each super state as a dfa state. If both the transition function and the encoding for super states are developed, then the dfa is constructed as follows:

```
(make-dfa <encoding of super states>
          Σ
          <encoding of E(s)>
          <encoding of super states that contain f∈F>
          <transition function between encoded super states>)
```

Fig. 27 An `ndfa` for `L = (aba ∪ ab)*`

```
(define ND
  (make-ndfa
    '(S A B C D E)
    '(a b)
    'S
    '(S)
    `((S a A)
      (S a B)
      (A b C)
      (B b D)
      (C a E)
      (D ,EMP S)
      (E ,EMP S))))
```

(a) Implementation.



(b) Transition diagram.

To compute the transition function, each known super state, `P`, must be processed. At the beginning, the only known super state is `E(s)`. For each state, `p`∈`P`, each alphabet element, `a`, is processed to compute, possibly new, super states. The union of the super states obtained from processing `a` for each `p`∈`P` is the super state the `dfa` moves to from `P` on an `a`. For instance, let `P` = {p_1 p_2 p_3}, and let `(`p_1 `a r),(`p_1 `a s),(`p_3 `a t)`$\in\delta$. On an `a`, from p_1, `N` may transition to any state in `E(r)`∪`E(s)`; from p_2, `N` may transition nowhere; and from p_3, `N` may transition to any state in `E(t)`. Therefore, we may describe a transition in the `dfa` as follows:

`((`p_1 p_2 p_3`) a (E(r) ∪ E(s)∪ E(t)))`

That is, the `dfa` transitions from super state `P` on an `a` to a super state `Q`, where `Q = (E(r) ∪ E(s) ∪ E(t))`.

Once the transition function between super states has been computed, it is a straightforward matter to extract the super states and generate `FSM` symbols to encode them. This encoding is then used to create the states, the starting state, the final states, and the rules for use with `make-dfa`.

To illustrate the proposed constructor, consider the `ndfa` displayed in Fig. 27 (in the interest of brevity, tests are omitted in Fig. 27a). First, the empties for each state are computed. The following table displays the results:

State	E(state)
S	(S)
A	(A)
B	(B)
C	(C)
D	(D S)
E	(E S)

Use the transition diagram displayed in Fig. 27b to verify the empties of each state are properly computed.

Second, the super state transition function is computed. It is helpful to use the transition diagram for this step. The process starts with the only known super state, E(S), which is the start super state for the dfa. For each element, a, of the alphabet, take the union of the states that can be reached from the states in E(S) by first consuming a. E(S) only contains S, and, thus, the following transitions are obtained:

```
((S) a (A B))
((S) b ())
```

Observe that two new needed super states have been discovered. The process is repeated for each unprocessed super state. Let us continue with (A B). Both A and B can go nowhere on an a. On a b, from A, the machine can transition to C, and from B, the machine can transition to D and by an ϵ-transition to S. The needed super state transitions are:

```
((A B) a ())
((A B) b (C D S))
```

Observe that there are two unprocessed super states: () and (C D S). Let us continue with the first. From the empty super state, only itself may be reached. Thus, the needed transitions are:

```
(() a ())
(() b ())
```

The process continues with the (C D S) super state. From C after consuming an a, the machine can transition to E or S. In D, a cannot be consumed, and, therefore, nothing is reachable. From S after consuming an a, both A and B are reachable. From C, D, and S on a b, nothing is reachable. The needed dfa transitions are:

```
((C D S) a (E S A B))
((C D S) b ())
```

The process continues with (E S A B) super state. From E on an a, nothing is reachable. From S after consuming an a, we have that A and B are reachable. Both A and B can go nowhere on an a. On a b, nothing is reachable from E and S, C is reachable from A, and D and S are reachable from B. The needed super state dfa transitions are:

```
((E S A B) a (A B))
((E S A B) b (C D S))
```

There are no more super states to process. This means that the super state transition function has been computed. The following table summarizes the super state transition function:

Super State	a	b
(S)	(A B)	()
(A B)	()	(C D S)
(C D S)	(E S A B)	()
(E S A B)	(A B)	(C D S)
()	()	()

To build a `dfa`, the super states must be mapped to `FSM` states. The following table is one such encoding:

Super state	dfa state
(S)	S
(A B)	A
(C D S)	B
(E S A B)	C
()	DEAD

Observe that `S`, `B`, and `C` are final states in the `dfa` because the super state they represent contains the only final state in `ND`.

The work done allows for the construction of the `dfa` displayed in Fig. 28. The tests validate that the deterministic and nondeterministic implementations decide the same language. The second `check-equal?` uses `FSM`'s primitive, `ndfa->dfa`, to transform an `ndfa` to a `dfa` and validates that `FSM`'s transformation and the `dfa` developed are equivalent.

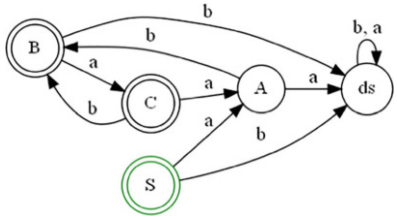
28.2 Implementation

To implement the proposed constructor, clear data definitions for the information manipulated are needed. The implementation presented uses the following data definitions:

Fig. 28 A dfa for L = (aba ∪ ab)*

```
(define D
  (make-dfa `(S A B C ,DEAD)
            '(a b)
            'S
            '(S B C)
            `((S a A)
              (S b ,DEAD)
              (A a ,DEAD)
              (A b B)
              (B a C)
              (B b ,DEAD)
              (C a A)
              (C b B)
              (,DEAD a ,DEAD)
              (,DEAD b ,DEAD))))

;; Tests for D
(check-equal?
  (sm-testequiv? D ND 500)
  #t)
(check-equal?
  (sm-testequiv? (ndfa->dfa ND)
                 D
                 500)
  #t)
```

(a) Implementation.



(b) Transition diagram.

```
;; Data Definitions
;;
;;  An ndfa transition rule, ndfa-rule, is a
;;  (list state symbol state)
;;
;;  A super state, ss, is a (listof state)
;;
;;  A super state dfa rule, ss-dfa-rule, is a
;;  (list ss symbol ss)
;;
;;  An empties table, emps-tbl, is a
;;  (listof (list state ss))
;;
;;  A super state name table, ss-name-table, is a
;;  (listof (list ss state))
```

Fig. 29 The main function to convert an `ndfa` to a `dfa`

```
;; ndfa → dfa
;; Purpose: Convert the given ndfa to an equivalent dfa
(define (ndfa2dfa M)
  (if (eq? (sm-type M) 'dfa)
      M
      (convert (sm-states M)
               (sm-sigma M)
               (sm-start M)
               (sm-finals M)
               (sm-rules M))))

;; Tests for ndfa2dfa
(define M (ndfa2dfa AT-LEAST-ONE-MISSING))
(check-equal? (sm-testequiv? AT-LEAST-ONE-MISSING M 500) #t)
(check-equal? (sm-testequiv? M (ndfa->dfa AT-LEAST-ONE-MISSING) 500) #t)

(define N (ndfa2dfa ND))
(check-equal? (sm-testequiv? ND N 500) #t)
(check-equal? (sm-testequiv? N (ndfa->dfa ND) 500) #t)
```

28.2.1 Main Function: `ndfa2dfa`

The main function, `ndfa2dfa`, tests if the given machine is a `dfa`. If so, it returns it given that no transformation is needed. Otherwise, a converting function is called with the components of the given `ndfa`. Following the steps of the design recipe for functions yields the code displayed in Fig. 29.

To test the function, two (previously defined) `ndfa`s are converted to `dfa`s. The `dfa`s are tested for equivalence with their nondeterministic counterparts and with the `dfa`s obtained using `FSM`'s `ndfa->dfa`.

28.2.2 The `convert` Function

The `convert` function takes as input the components of an `ndfa`. To build a `dfa`, it computes the following values:

1. The empties table
2. The super state `dfa` rules
3. The super state name table

An auxiliary function, `compute-empties-tbl`, that takes as input the given states and the given rules is used to compute the empties table.

Fig. 30 The convert implementation

```
;; (listof state) alphabet state (listof state) (listof ndfa-rule) → dfa
;; Purpose: Create a dfa from the given ndfa components
(define (convert states sigma start finals rules)
  (let* [(empties (compute-empties-tbl states rules))
         (ss-dfa-rules
          (compute-ss-dfa-rules (list (extract-empties start empties))
                                sigma
                                empties
                                rules
                                '()))
         (super-states (remove-duplicates
                        (append-map (λ (r) (list (first r) (third r)))
                                    ss-dfa-rules)))
         (ss-name-tbl (compute-ss-name-tbl super-states))]
    (make-dfa (map (λ (ss) (second (assoc ss ss-name-tbl)))
                   super-states)
              sigma
              (second (assoc (first super-states) ss-name-tbl))
              (map (λ (ss) (second (assoc ss ss-name-tbl)))
                   (filter (λ (ss) (ormap (λ (s) (member s finals)) ss))
                           super-states))
              (map (λ (r) (list (second (assoc (first r) ss-name-tbl))
                                (second r)
                                (second (assoc (third r) ss-name-tbl))))
                   ss-dfa-rules)
              'no-dead)))
;; Tests for convert
(check-equal?
  (sm-testequiv? (convert '(S A B) '(a b) 'S '(A B) '((S a A)
                                                      (S a B)
                                                      (A a A)
                                                      (B b B)))
                 (make-ndfa '(S A B) '(a b) 'S '(A B) '((S a A)
                                                        (S a B)
                                                        (A a A)
                                                        (B b B)))
                 500)
  #t)
(check-equal?
  (sm-testequiv? (convert '(S A) '(a b) 'S '(S A) '((S a S)
                                                    (S a A)
                                                    (A b A)
                                                    (A a A)))
                 (make-ndfa '(S A) '(a b) 'S '(S A) '((S a S)
                                                      (S a A)
                                                      (A b A)
                                                      (A a A)))
                 500)
 #t)
```

Computing the super state `dfa` rules is also delegated to an auxiliary function. The auxiliary function performs a breadth-first search using two accumulators, the given alphabet, the empties table, and the given rules. The first accumulator is for the super states that must still be visited. Initially, this accumulator only contains one super state: the empties of the given starting state. This value is extracted from the empties table entry for the given state using the following function:

```
;; state emps-tbl → ss
;; Purpose: Extract the empties of the given state
;; Assume: Given state is in the given list of states
(define (extract-empties st empties)
  (second (first (filter (λ (e) (eq? (first e) st))
                         empties))))

;; Tests for extract-empties
(check-equal? (extract-empties 'A '((S (S B))
                                    (F (F))
                                    (A (A C D))
                                    (C (C))
                                    (D (D))))
              '(A C D))

(check-equal? (extract-empties 'Z '((Z (Z S))
                                    (S ())))
              '(Z S))
```

The second accumulator is for the already visited super states. Initially, this accumulator is the empty list. Given that the first super state processed is the one for `start`, the first super state `dfa` rule is a rule for the starting super state.

Computing the super state name table requires processing the super states and is delegated to an auxiliary function. Given that the super states are needed several times to construct the `dfa`, these are locally defined. They are computed by extracting all instances of a super state in the super state `dfa` rules and removing duplicates. Given that the first rule is from the starting super state, the first state in the list of super states is the starting super state.

The `dfa` is constructed with the following arguments for the constructor:

States: The list of states obtained from removing duplicates from the mapping of all instances of a super state in the super state `dfa` rules to a state using the super state name table.

Alphabet: The given alphabet.

Start State: The mapping of the starting super state (the first in the list of super states) to its `dfa` state using the super state name table.

Final States: The mapping of all super states that contain a given final state to their `dfa` states using the super state name table.

Fig. 31 The function to compute the empties table

```
;; (listof state) rules → emps-tbl
;; Purpose: Compute empties table for all given states
(define (compute-empties-tbl states rules)
  ;; state (listof state) (listof ndfa-rule) → (listof ndfa-rule)
  ;; Purpose: Extract empty transitions to non-generated states for the
  ;;          given state
  (define (get-e-trans state gen-states rules)
    (filter (λ (r) (and (eq? (first r) state)
                        (eq? (second r) EMP)
                        (not (member (third r) gen-states))))
            rules))
  ;; (listof state) (listof ndfa-rules) (listof state) → (listof state)
  ;; Purpose: Compute the empties for the states left to explore in the first
  ;;          given (listof state)
  ;; Accumulator Invariants:
  ;;     to-search = unvisited states reachable by consuming no input
  ;;       visited = visited states reachable by consuming no input
  (define (compute-empties to-search rules visited)
    (if (empty? to-search)
        visited
        (let* [(curr (first to-search))
               (curr-e-rules
                 (get-e-trans curr (append to-search visited) rules))]
          (compute-empties (append (rest to-search) (map third curr-e-rules))
                           rules
                           (cons curr visited)))))
  (map (λ (st) (list st (compute-empties (list st) rules '()))) states))
;; Tests for compute-empties-tbl
(check-equal? (compute-empties-tbl '(X Y Z) `((X ,EMP Y) (Y a Z) (Z ,EMP X)))
              '((X (Y X)) (Y (Y)) (Z (Y X Z))))
(check-equal?
  (compute-empties-tbl '(W X Y Z)
                       `((W ,EMP X) (X ,EMP Y) (Y a Z) (Z ,EMP Y) (Z b Z)))
  '((W (Y X W)) (X (Y X)) (Y (Y)) (Z (Y Z))))
```

Transition Function: The super state `dfa` rules with each state mapped to their `dfa` state using the super state name table.

'no-dead: This indicates to `make-dfa` not to add a dead state. This is safe to do because `ss-dfa-rules` is a function between super states.

The result of this design is displayed in Fig. 30. The tests are written to validate the equivalence of an `ndfa` and the conversion obtained from its components. The `assoc` function takes as input a value (e.g., a super state) and a table (e.g., the super state name table) and returns the table entry whose first element matches the given value.

28.2.3 Computing the Empties Table

To compute the empties table, the set of states and the transition rules of an `ndfa` are received as input. The table is created by mapping a function that creates a table entry for the given states. The mapped function computes the empties for a state. The computation of the empties for a state is delegated to an auxiliary function that performs a breadth-first search through the given rules and consumes the accumulator value for the states to visit, the given rules, and the accumulator value for the visited states. Initially, the first accumulator only contains the state being processed, and the second accumulator value is empty.

The function to compute the empties of state, `compute-empties`, is designed using generative recursion with two accumulators to perform a breadth-first search for states through ϵ-transitions. If there are no more states to visit, then the accumulator for the visited states contains all the states reachable by ϵ-transitions and is returned. Otherwise, the first unvisited state's ϵ-transitions to states that are not in the visited list nor the unvisited list are extracted from the given rules. In this manner, a state is never visited more than once. The search recursively continues with the list obtained by adding the third element of the extracted rules to the rest of the states to visit, the given rules, and the list obtained by adding the first unvisited state to the accumulator for the visited states.

To extract the needed ϵ-transitions, `get-e-trans`, consumes the state to process, the list of generated states (i.e., the states to visit and already visited), and the rules to search. It filters the list of states to return the rules that go from the given state consuming no input to a state that has not been generated.

The result of this design is displayed in Fig. 31. The tests are purposely left small enough to easily illustrate to any reader of the code the expected behavior of the function.

28.2.4 Computing the Super State Transition Function

To compute the super state transition function, a breadth-first search rooted at the super states that still need to be explored is performed to identify transition rules. There are two accumulators: one for the super states left to explore and one for the super states already explored. If there are no states left to explore, the empty list is returned because the search for super state `dfa` rules is done. Otherwise, the first unexplored super state, `curr-ss`, is pro-

Fig. 32 Function to compute the super state transition function

```
;; (listof ss) alphabet emps-tbl (listof ndfa-rule) (listof ss)
;;                                                  → (listof ss-dfa-rule)
;; Purpose: Compute the super state dfa rules
;; Accumulator Invariants:
;;               ssts = the super states explored
;;     to-search-ssts = the super states that must still be explored
(define (compute-ss-dfa-rules to-search-ssts sigma empties rules ssts)
      ⋮
 (if (empty? to-search-ssts)
     '()
     (let* [(curr-ss (first to-search-ssts))
            (reachables (find-reachables curr-ss sigma rules empties))
            (to-super-states
             (build-list (length sigma) (λ (i) (get-reachable i reachables))))
            (new-rules (map (λ (sst a) (list curr-ss a sst))
                            to-super-states
                            sigma))]
       (append
         new-rules
         (compute-ss-dfa-rules
           (append (rest to-search-ssts)
                   (filter (λ (ss)
                             (not (member ss (append to-search-ssts ssts))))
                           to-super-states))
           sigma
           empties
           rules
           (cons curr-ss ssts))))))
```

cessed. The set of reachable states for every state in the `curr-ss`, consuming every element of the alphabet, is computed. For instance, if the `curr-ss` is `'(A B C)` and the alphabet is `'(a b)`, then the reachable states have the following structure:

```
(((reachable from A on a) (reachable from A on b))
 ((reachable from B on a) (reachable from B on b))
 ((reachable from C on a) (reachable from C on b)))
```

There are three sublists in the list: one for each state in `curr-ss`. Each sublist has a list of states that are reachable for each element of the alphabet. From this, the super states to transition into, `to-super-states`, are computed. For instance, the super state reachable on an `a` is formed by all the states reachable on an `a` from `A`, `B`, and `C` (without repetitions). The `dfa` transition rules are generated by simultaneously traversing `to-super-states` and the alphabet. Assuming `a` is the current alphabet element and `sst` is the current element of `to-super-states` during such a traversal, then a super state `dfa` rule of the following form is created: `(curr-ss a sst)`. Finally, the super state `dfa` rules generated are appended with the result of recursively pro-

cessing a new accumulator for unexplored super states containing the rest of the said accumulator and any new super states generated for the new rules, the same alphabet, the same empties table, the same list of `ndfa` rules, and a new accumulator for explored states obtained by adding `curr-ss`. The outline of this design is displayed in Fig. 32. This is the bulk of what is needed to compute the super state `dfa` transition function. We proceed next to discuss the implementation of the auxiliary functions (that ought to go where the vertical dots are in Fig. 32).

To find the reachable super states from a given super state, a function is mapped onto the super state to process its states. The mapped function takes as input a state and computes the reachable super states from the given state using the given alphabet, rules, and empties table. It is implemented as follows:

```
;; ss alphabet (listof ndfa-rule) emps-tbl
;;                                    → (listof (listof ss))
;; Purpose: Compute reachable super states from given
;;          super state
(define (find-reachables ss sigma rules empties)
  (map (λ (st)
         (find-reachables-from-st st sigma rules empties))
       ss))
```

The auxiliary function to find the reachable super states from a state maps a function onto the alphabet. This function takes as input an alphabet element and calls an auxiliary function to find the reachable super state from the given state and the given alphabet element. It is implemented as follows:

```
;; state alphabet (listof ndfa-rule) emps-tbl → (listof ss)
;; Purpose: Find the reachable super state from the given state
;;          for each element of the given alphabet
(define (find-reachables-from-st st sigma rules empties)
  (map (λ (a)
         (find-reachables-from-st-on-a st a rules empties))
       sigma))
```

The function to find the reachable super states from a given state and a given alphabet element extracts all the rules for the given state on the given alphabet element. The third element of these rules are all the reachable states by consuming the single alphabet element. The empties of each of these states are appended, and duplicates are removed to obtain the reachable super state. The function is implemented as follows:

```
;; state symbol (listof ndfa-rule) emps-tbl → ss
;; Purpose: Find the reachable super state from the given state
;;          and the given alphabet element
 (define (find-reachables-from-st-on-a st a rules empties)
   (let* [(rls (filter
                (λ (r)
                  (and (eq? (first r) st) (eq? (second r) a)))
                rules))
          (to-states (map third rls))]
     (remove-duplicates
       (append-map (λ (st) (extract-empties st empties))
                   to-states))))
```

Finally, to construct a super state from the `ith` super state in each sublist of a `(listof (listof super-state))`, each sublist is referenced at `i`, and the results are appended. The needed super state is obtained by removing duplicates. The function is implemented as follows:

```
;; natnum (listof (listof ss)) → (listof ss)
;; Purpose: Return ss of ith (listof state) in each given
;;          list element
(define (get-reachable i reachables)
  (remove-duplicates (append-map
                      (λ (reached) (list-ref reached i))
                      reachables)))
```

28.2.5 Computing the Super State Name Table

To create the super state name table, a function is mapped to the given list of super states. This function creates a table entry for the given super state. If the given super state is empty, then its entry contains `FSM`'s `DEAD` state. For a nonempty super state, an `FSM` symbol is generated. The function is implemented as follows:

```
;; (listof ss) → ss-name-tbl
;; Purpose: Create a table for ss names
(define (compute-ss-name-tbl super-states)
  (map (λ (ss)
         (if (empty? ss)
             (list ss DEAD)
             (list ss (generate-symbol 'Z '(Z)))))
       super-states))
```

```
;; Tests for compute-ss-name-tbl
(check-pred (lambda (tbl)
              (and (list? tbl)
                   (andmap (λ (e) (= (length e) 2)) tbl)
                   (andmap (λ (e) (andmap symbol? (first e)))
                           tbl)
                   (andmap (λ (e) (symbol? (second e))) tbl)))
            (compute-ss-name-tbl '()))

(check-pred (lambda (tbl)
              (and (list? tbl)
                   (andmap (λ (e) (= (length e) 2)) tbl)
                   (andmap (λ (e) (andmap symbol? (first e)))
                           tbl)
                   (andmap (λ (e) (symbol? (second e))) tbl)))
            (compute-ss-name-tbl '((A B) (A B C) () (C))))
```

Given that `FSM` symbol generation is done randomly, model checking is used to test the function. Each table generated must be a list of entries. Each entry must be a list of length 2. In addition, each entry must have a super state as its first element and a symbol as its second element.

This completes the design and implementation of the function to convert an `ndfa` to an equivalent `dfa`. Run the tests and make sure they all pass.

28.3 Correctness Proof

After validating the construction algorithm through testing, it is necessary to prove that the algorithm is correct. To prove algorithm correctness, assume all functions work properly. Based on this assumption, we need to prove that the language of the given `ndfa` is the same as the language of the computed `dfa`. The proof is tightly coupled with the construction of the `dfa`. More formally, let:

```
ND = (make-ndfa S Σ A F δ)

 D = (make-dfa S' Σ A' F' δ'). where
       S' = the states computed in convert
       A' = the starting state computed in convert
       F' = the final states computed in convert
       δ' = the transition function computed in convert
```

We need to prove `L(ND) = L(D)`.

28.3.1 Proof That D Simulates All ND Computations and Vice Versa

To prove that the languages are the same, we first need to prove that every computation possible with ND may be carried out by D and vice versa. To this end, we prove the following theorem by induction on |w|.

Theorem 3 *(w Q)* $\vdash^*_{ND}$ *(() P)* $\Leftrightarrow$ *(w Q')* $\vdash^*_{D}$ *(() P'), where Q' = E(Q)* $\wedge$ *P*$\in$*P'.*

Proof
($\Rightarrow$) Assume (w Q) $\vdash^*_{ND}$ (() P).

Base Case: |w| = 0
|w| = 0 $\Rightarrow$ w = ()

By assumption, (() Q) $\vdash^*_{ND}$ (() P). This means that P$\in$E(Q).

By construction of D, P$\in$Q'. This means that ND's computation is carried out as follows by D:

(() Q') $\vdash$ (() Q')

This establishes the base case.

Inductive Step:

Assume: (w Q) $\vdash^*_{ND}$ (() P) $\Rightarrow$ (w Q') $\vdash^*_{D}$ (() P'), where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k.
Show: (w Q) $\vdash^*_{ND}$ (() P) $\Rightarrow$ (w Q') $\vdash^*_{D}$ (() P'), where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k+1.

|w|=k+1 $\Rightarrow$ w=(xa), where x$\in$ Σ^* and a$\in$ Σ.

To prove the implication, assume ((xa) Q) $\vdash^*_{ND}$ (() P).

This means that ND's computation is:

(xa Q) $\vdash^*_{ND}$ ((a) R) $\vdash_{ND}$ (() T) $\vdash^*_{ND}$ (() P)

That is, consuming x takes ND from Q to some intermediate state R. From R on an a, ND goes to T. Then by ϵ-transitions, ND gets to P.

By inductive hypothesis:

((xa) Q') $\vdash^*_D$ ((a) R'), where R$\in$R'.

Observe that (R a T) $\in \delta$ and P $\in$ E(T). Let P' be E(T). By construction of D, this means that the following is D's computation:

((xa) Q') $\vdash^*_D$ ((a) R') $\vdash_D$ (() P'), where P$\in$P'.

Clearly, we have that (w Q') $\vdash^*_D$ (() P'), where P$\in$P'. This completes the proof of the implication.

($\Leftarrow$) Assume: (w Q') $\vdash^*_D$ (() P'), where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k+1.

Base Case: |w| = 0
|w| = 0 $\Rightarrow$ w = ()

By assumption, Q' = P' because D is deterministic. This means that P$\in$E(Q).

By construction of D:

(() Q) $\vdash^*_{ND}$ (() P)

This establishes the base case.

Inductive Step:

Assume: (w Q') $\vdash^*_D$ (() P') $\Rightarrow$ (w Q) $\vdash^*_{ND}$ (() P),
where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k.
Show: (w Q') $\vdash^*_D$ (() P') $\Rightarrow$ (w Q) $\vdash^*_{ND}$ (() P),
where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k+1.

|w|=k+1 $\Rightarrow$ w=(xa), where x$\in \Sigma^*$ and a$\in \Sigma$.

To prove the implication, assume (w Q') $\vdash^*_D$ (() P'), where Q' = E(Q) $\wedge$ P$\in$P', for |w|=k+1.

This means that D's computation is:

((xa) Q') $\vdash^*_D$ ((a) R') $\vdash_D$ (() P')

By construction of D, the above means:

((xa) Q) $\vdash^*_{ND}$ ((a) R) $\vdash_{ND}$ (() T) $\vdash^*_{ND}$ (() P), where R∈R' and P∈E(T)=P'.
This completes the proof of the theorem. □

We can now prove that the two machines decide the same language.

28.3.2 `L(ND)` = `L(D)` Proof

As ought to be expected by now, the proof that `L(ND)` = `L(D)` is divided into two lemmas: one for words that are in the language of both machines and another for words that are not in the language of both machines.

Lemma 3 *w∈L(ND) ⇔ w∈L(D)*

Proof (⇒) Assume w∈L(ND).

This means that:

(w S) $\vdash^*_{ND}$ (() P), where P∈F.

By Theorem 3, we have that:

(w S') $\vdash^*_{D}$ (() P'), where P∈P'.

By construction of D, P'∈F'. Thus, w∈L(D).

(⇐) Assume w∈L(D).

This means that:

(w S') $\vdash^*_{D}$ (() P'), where P'∈F'.

By Theorem 3 and construction of D, we have that:

(w S) $\vdash^*_{ND}$ (() P), where P∈F and P∈P'.

Thus, w∈L(ND). □

Lemma 4 *w∉L(ND) ⇔ w∉L(D)*

Proof (⇒) Assume w∉L(ND).

This means that for all ND computations:

$$(\mathrm{w\ S}) \vdash^*_{ND} ((\,)\ \mathrm{P}), \text{ where } \mathrm{P} \notin \mathrm{F}.$$

By Theorem 3, we have that:

$$(\mathrm{w\ S'}) \vdash^*_{D} ((\,)\ \mathrm{P'}), \text{ where } \mathrm{P} \in \mathrm{P'}.$$

By construction of D, $\mathrm{P'} \notin \mathrm{F'}$. Thus, $\mathrm{w} \notin \mathrm{L(D)}$.

($\Leftarrow$) Assume $\mathrm{w} \notin \mathrm{L(D)}$.

This means that:

$$(\mathrm{w\ S'}) \vdash^*_{D} ((\,)\ \mathrm{P'}), \text{ where } \mathrm{P'} \notin \mathrm{F'}.$$

By Theorem 3 and construction of D, we have that:

$$(\mathrm{w\ S}) \vdash^*_{ND} ((\,)\ \mathrm{P}), \text{ where } \mathrm{P} \notin \mathrm{F} \text{ and } \mathrm{P} \in \mathrm{P'}.$$

Thus, $\mathrm{w} \notin \mathrm{L(ND)}$. □

The two lemmas lead to the sought theorem:

Theorem 4 *L(ND) = L(D)*

Proof
Lemmas 3 and 4 establish the theorem. □

29 Concluding Remarks

It is a remarkable result that endowing `dfa`s with nondeterminism yields no extra computational power. At its beginning, this chapter states that it is difficult to see how the following language can be decided by a `dfa`:

$$\mathtt{L} = \mathtt{ab}^* \cup \mathtt{aa}^* \cup \epsilon$$

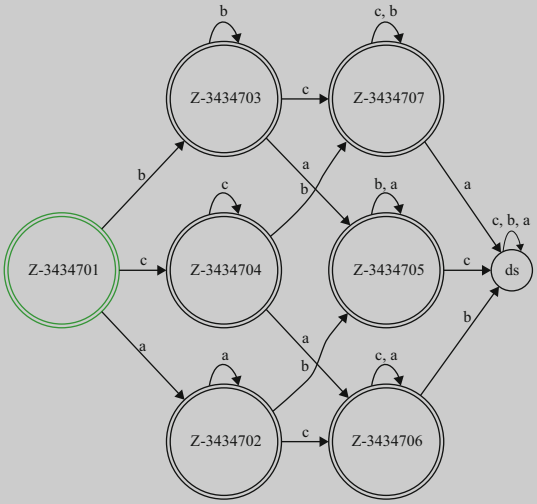
Indeed, it was not easy to see. Now, however, it ought to be clear that there is a `dfa` that can decide `L`. To obtain such a `dfa`, design an `ndfa` for `L`, and convert it to a `dfa`.

Does this mean that `ndfa`s are worthless? For a computer scientist, the answer is clearly no. Although `ndfa`s do not provide more computational power, they do make the design process easier. That is, they are a useful abstraction programmers may use to design solutions using a `dfa`. You may ponder about this as follows. Can everything you do with your favorite higher-level

programming language be done in assembly? Clearly, the answer is yes. That is, the higher-level programming language does not offer any additional computational power. Nonetheless, you choose not to program in assembly and prefer a higher-level language. Why? The higher-level programming language offers abstractions that make problem-solving and programming easier. Such is the case for `ndfa`s.

6 Prove that any `ndfa` may be implemented with a single final state.

7 Converting `AT-LEAST-ONE-MISSING` to a `dfa` yields a machine, `DM`, with the following transition diagram:



Prove that `L(AT-LEAST-ONE-MISSING)` = `L(DM)`.

8 Attempt to directly prove by induction on |w| that `L(ND)` = `L(D)`. What goes wrong with this approach?

Chapter 7
Finite-State Automatons and Regular Expressions

Our study of finite-state automatons, deterministic or nondeterministic, has suggested that the languages they decide are the same as the regular languages. We have seen `dfa` examples that read concatenated symbols or that loop to read a collection of concatenated symbols an arbitrary number of times. This suggests that the languages they decide may be closed under concatenation and Kleene star. We have seen `ndfa` examples that suggest the languages they decide are closed under union. It is striking that these are operations used by regular expressions to define a language.

30 Closure Properties

If the languages decided by finite-state automatons are the regular languages, then they must be closed under concatenation, union, and Kleene star. That is, we ought to be able to combine the languages decided by finite-state automatons, like regular expressions are combined, using concatenation, union, and Kleene star to create machines for bigger languages.

Such an ability would provide programmers with a new set of constructors to create finite-state automatons, thus, simplifying the amount of work a programmer must do to create "complex" finite-state automatons. We shall prove the following theorem establishing closure properties for the languages decided by finite-state automatons:

Theorem 1 *The languages decided by finite-state automatons are closed under:*

(a) Union
(b) Concatenation
(c) Kleene star
(d) Complement
(e) Intersection

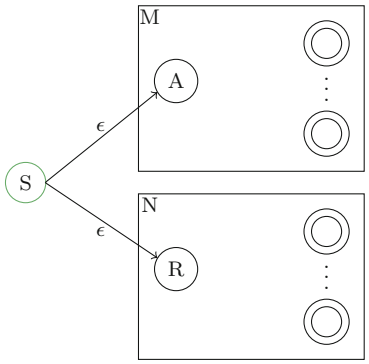
M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_7

Proof The proof is divided into five theorems, Theorems 2 to 6, below. They are all proven using a constructive proof. □

To test the constructors, the following machines are defined:

```
;; L = ab*
(define ab* (make-ndfa '(S A)
                       '(a b)
                       'S
                       '(A)
                       '((S a A)
                         (A b A))))

;; L = a(a U ab)b*
(define a-aUb-b* (make-ndfa '(Z H B C D F)
                            '(a b)
                            'Z
                            '(F)
                            `((Z a H)
                              (Z a B)
                              (H a D)
                              (D ,EMP F)
                              (B a C)
                              (C b F)
                              (F b F))))

;; L = aab*
(define aab* (make-ndfa '(W X Y)
                        '(a b)
                        'W
                        '(Y)
                        '((W a X)
                          (X a Y)
                          (Y b Y))))

;; L = a*
(define a* (make-dfa '(S D)
                     '(a b)
                     'S
                     '(S)
                     '((S a S)
                       (S b D)
                       (D a D)
                       (D b D))
                     'no-dead))
```

Fig. 33 Construction algorithm for L = L(M) ∪ L(N)



30.1 Union

30.1.1 Construction Algorithm

Let the following be the two machines that decide the languages to union:

```
M = (make-ndfa S_M Σ_M A F_M δ_M)
N = (make-ndfa S_N Σ_N R F_N δ_N)
```

We need to construct an `ndfa` that decides `L = L(M) ∪ L(N)`. The construction algorithm may be visualized as displayed in Fig. 33. Without loss of generality, assume that `(sm-states M) ∩ (sm-states N) = ∅`. A starting state, `S`, for the union machine is created such that it is not in `(sm-states M)` nor in `(sm-states N)`. The union machine nondeterministically decides to go to `M`'s starting state or `N`'s starting state. After that, the union machine simulates the machine whose starting state it moved to. Given that either `M` or `N` may be simulated, the union machine needs the alphabets, the final states, and the rules of both.

The construction algorithm is implemented as displayed in Fig. 34. It assumes that the given machines do not have a state with the same name. The components of the union machine are computed first. A new start state is generated such that it is not a state in the union of the states of the given machines. The union's machine alphabet is computed by appending the alphabets of both given machines and removing duplicates. The states of the new machine are obtained by adding the new starting state to the union of the states of both given machines. The union machine's final states contains the final states of the given machines. The rules for the union machine include the rules of both given machines and the empty transition rules from the new start state to the start states of the given machines.

Fig. 34 The implementation of the union of two `ndfa`s

```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of the languages of the
;;          given ndfas
;; Assume: The intersection of the states of the given machines is empty
(define (union-fsa M N)
  (let* [(new-start (generate-symbol
                     'S (append (sm-states M) (sm-states N))))
         (new-sigma (remove-duplicates
                     (append (sm-sigma M) (sm-sigma N))))
         (new-states (cons new-start
                           (append (sm-states M) (sm-states N))))
         (new-finals (append (sm-finals M) (sm-finals N)))
         (new-rules (append (list (list new-start EMP (sm-start M))
                                  (list new-start EMP (sm-start N)))
                            (sm-rules M)
                            (sm-rules N)))]
    (make-ndfa new-states new-sigma new-start new-finals new-rules)))

;; Tests for union-fsa
(define ab*Ua-aUb-b* (union-fsa ab* a-aUb-b*))
(define ab*Uaab* (union-fsa ab* aab*))

(check-equal? (sm-apply ab*Ua-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a a)) 'reject)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a b)) 'accept)
(check-equal? (sm-apply ab*Ua-aUb-b* '(a a b b)) 'accept)
(check-equal? (sm-testequiv? ab*Ua-aUb-b* (sm-union ab* ab*Ua-aUb-b*))
              #t)

(check-equal? (sm-apply ab*Uaab* '(a a a)) 'reject)
(check-equal? (sm-apply ab*Uaab* '(b a b a)) 'reject)
(check-equal? (sm-apply ab*Uaab* '(a b b)) 'accept)
(check-equal? (sm-apply ab*Uaab* '(a a b)) 'accept)
(check-equal? (sm-apply ab*Uaab* '(a b b b b)) 'accept)
(check-equal? (sm-testequiv? ab*Uaab* (sm-union ab* aab*)) #t)
```

For testing, two `ndfa`s are created using the new constructor. The constructor is tested by illustrating the expected behavior using words that are not and that are in `L(M)` ∪ `L(N)`. In addition, the equivalence of the created union machines and the machines obtained using `FSM`'s `sm-union` is tested.

30.1.2 Algorithm Correctness Proof

Define three machines as follows:

`M = (make-ndfa S` Σ `Z F` δ`)`
`N = (make-ndfa S' Σ' Z' F'` δ`')`
`U = (union-fsa M N) = (make-ndfa S'' Σ'' Z'' F''` δ`'')`

Let `L = L(M) ∪ L(N)`. We proceed to prove that `L = L(U)`.

Lemma 1 *w∈L ⇔ w∈L(U)*

Proof

(⇒) We need to show that w∈L ⇒ w∈L(U).

Assume w∈L. This means that w∈L(M) or w∈L(N). By construction, U non-deterministically correctly chooses to simulate M or to simulate N. Thus, w∈L(U).

(⇐) We need to show that w∈L(U) ⇒ w∈L.

Assume w∈L(U). This means that there is a computation that consumes w such that:

$$(w\ Z'') \vdash^*_U (()\ K), \text{ where } K \in F''$$

By U's construction, either M or N is simulated, and U's final states are the final states of M and N. This means w∈L(M) or w∈L(N). Thus, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(U)*

Proof

(⇒) We need to show that w∉L ⇒ w∉L(U).

Assume w∉L. This means that w∉L(M) and w∉L(N). By U's construction, $F'' = F \cup F'$, and all possible computations of U on w never reach a state in F''. Thus, w∉L(U).

(⇐) We need to show that w∉L(U) ⇒ w∉L.

Assume w∉L(U). This means that all possible computations on w are described as follows:

$$(w\ Z'') \vdash^*_U (()\ K), \text{ where } K \notin F''$$

By U's construction, either M or N is simulated, and $F'' = F \cup F'$. This means w∉L(M) and w∉L(N). Thus, w∉L. □

Theorem 2 *The languages accepted by finite-state machines are closed under union.*

Proof The theorem is established by Lemmas 1 and 2. □

1 The union constructor assumes that the intersection of the states of the given machines is empty. Why is this assumption necessary? What may go wrong if this intersection is not empty? Carefully justify your answer.

2 If `M` is an `ndfa` that decides `L(M)` and `N` is an `ndfa` that decides `L(N)` and either has unreachable states, then `(union-fsa M N)` also has unreachable states. Redesign `union-fsa` so that the returned machine does not have unreachable states.

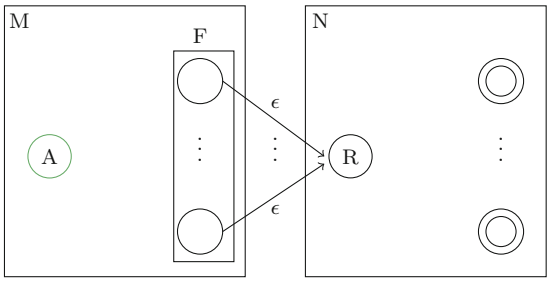
3 A famous hacker claims that the following is a more efficient implementation for `union-fsa`:

```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the union of the languages of
;;          the given ndfas
;; Assume: The intersection of the states of the given
;;         machines is empty
(define (union-fsa M N)
  (let* [(new-start (sm-start M))
         (new-sigma (remove-duplicates
                     (append (sm-sigma M) (sm-sigma N))))
         (new-states (append (sm-states M) (sm-states N)))
         (new-finals (append (sm-finals M) (sm-finals N)))
         (new-rules (cons (list new-start EMP (sm-start N))
                          (append (sm-rules M)
                          (sm-rules N))))]
    (make-ndfa new-states
               new-sigma
               new-start
               new-finals
               new-rules)))
```

Is the famous hacker correct? Carefully, justify your answer.

4 Define and implement a constructor that takes as input 3 `ndfa`s and returns a machine for the union of the languages decided by the given machines. Prove the correctness of your construction algorithm.

Fig. 35 Construction algorithm for L = L(M) ∘ L(N)



30.2 Concatenation

Let the following be the two machines that decide the languages to concatenate:

```
M = (make-ndfa S_M  Σ_M  A F_M  δ_M)
N = (make-ndfa S_N  Σ_N  R F_N  δ_N)
```

We need to construct an `ndfa` that decides `L = L(M) ∘ L(N)`. Every word must start with `w∈L(M)` and end with `x∈L(N)`. Without loss of generality, assume that $S_M \cap S_N = \varnothing$. The construction algorithm may be visualized as displayed in Fig. 35. The constructed machine first simulates `M`. From every final state in F_M, the constructed machine can reach the `R` through an ϵ-transition. From there, it simulates `N`. For this, the constructed machine needs $S_M \cup S_N$ as its sets of states, $\Sigma_M \cup \Sigma_N$ as its alphabet, `A` as its starting state, F_N as its set of final states, and all the rules of `M` and `N` along with the ϵ-transitions from every final state in F_M to `R` as its set of rules.

30.2.1 Implementation

The construction algorithm is implemented as displayed in Fig. 36. The components of the concatenation machine are computed first. The concatenation machine's alphabet is computed by appending the alphabets of both given machines and removing duplicates. The states of the new machine are obtained by appending the states of both given machines. The rules for the concatenation machine include the rules of both given machines and the new ϵ-transitions rules. The latter are computed by mapping a function to the first given machine's final states. The mapped function creates an `ndfa` rule using the given final state, `EMP`, and the starting state of the second given machine.

For testing, two `ndfa`s are created using the new constructor. The constructor is tested by illustrating the expected behavior using words that are

Fig. 36 The implementation of the concatenation of two ndfas

```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the concatenation of the languages of the
;;          given ndfas
;; Assume: The intersection of the states of the given machines is empty
(define (concat-fsa M N)
  (let* [(new-start (sm-start M))
         (new-sigma (remove-duplicates (append (sm-sigma M) (sm-sigma N))))
         (new-states (append (sm-states M) (sm-states N)))
         (new-finals (sm-finals N))
         (new-rules (append (sm-rules M)
                            (sm-rules N)
                            (map (λ (f) (list f EMP (sm-start N)))
                                 (sm-finals M))))]
    (make-ndfa new-states new-sigma new-start new-finals new-rules)))

;; Tests for concat-fsa
(define ab*-o-a-aUb-b* (concat-fsa ab* a-aUb-b*))
(define ab*-o-aab* (concat-fsa ab* aab*))

(check-equal? (sm-apply ab*-o-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(b b b)) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a a b a b)) 'reject)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b a a b)) 'accept)
(check-equal? (sm-apply ab*-o-a-aUb-b* '(a b b b a a)) 'accept)
(check-equal? (sm-testequiv? ab*-o-a-aUb-b* (sm-concat ab* a-aUb-b*)) #t)

(check-equal? (sm-apply ab*-o-aab* '()) 'reject)
(check-equal? (sm-apply ab*-o-aab* '(a b a)) 'reject)
(check-equal? (sm-apply ab*-o-aab* '(a a b b a a)) 'reject)
(check-equal? (sm-apply ab*-o-aab* '(a b b a a b b)) 'accept)
(check-equal? (sm-apply ab*-o-aab* '(a a a)) 'accept)
(check-equal? (sm-testequiv? ab*-o-aab* (sm-concat ab* aab*)) #t)
```

not and that are in L(M) ∘ L(N). In addition, the equivalence of the created concatenation machines and the machines obtained using FSM's sm-concat is tested.

30.2.2 Algorithm Correctness Proof

Define three machines as follows:

M = (make-ndfa S Σ Z F δ)
N = (make-ndfa S' Σ' Z' F' δ')
U = (concat-fsa M N) = (make-ndfa S'' Σ'' Z'' F'' δ'')

Let L = L(M) ∘ L(N). We proceed to prove that L = L(U).

Lemma 3 *w∈L ⇔ w∈L(U)*

Proof

(⇒) We need to show that w∈L ⇒ w∈L(U).

Assume w∈L. This means w = xy, where x∈L(M) and y∈L(N). By construction of U, the following is a valid computation:

(xy S'') $\vdash^*_U$ (y R) $\vdash$ (y Z') $\vdash^*_U$ (() T), where R∈F and T∈F'.

By construction of U, F'' = F'. Therefore, w∈L(U).

(⇐) We need to show that w∈L(U) ⇒ w∈L.

Assume w∈L(U). By construction of U, this means that for w = xy, the following computation is valid:

(xy S'') $\vdash^*_U$ (y R) $\vdash$ (y Z') $\vdash^*_U$ (() T), where R∈F and T∈F'.

This implies that x∈L(M) and y∈L(N). Thus, w∈L. □

Lemma 4 *w∉L ⇔ w∉L(U)*

Proof

(⇒) We need to show that w∉L ⇒ w∉L(U).

Assume w∉L. This means w ≠ xy, where x∈L(M) and y∈L(N). By construction, all possible computations of U on w either do not reach a final state by consuming w or do not consume all of w. In both cases, w is rejected. Therefore, w∉L(U).

(⇐) We need to show that w∉L(U) ⇒ w∉L.

Assume w∉L(U). By construction, this means that w is rejected because U consumes w and does not reach a final state or U is unable to consume w. This implies that w cannot be written as xy, where x∈L(M) and y∈L(N). Thus, w∉L. □

Theorem 3 *The languages accepted by finite-state machines are closed concatenation.*

Proof The theorem is established by Lemmas 3 and 4. □

5 The concatenation constructor assumes that the intersection of the states of the given machines is empty. Why is this assumption necessary? What may go wrong if this intersection is not empty? Carefully justify your answer.

6 If `M` is an `ndfa` that decides `L(M)` and `N` is an `ndfa` that decides `L(N)`, then:

```
L(M)=∅ ⇒ (concat-fsa M N) = N
L(N)=∅ ⇒ (concat-fsa M N) = M
```

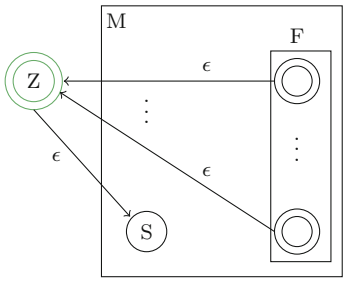
Redesign `concat-fsa` so that a new machine is not constructed when the language decided by either input machine is empty.

7 A famous computer science professor claims that `concat-fsa` may also be implemented as follows:

```
;; ndfa ndfa → ndfa
;; Purpose: Construct ndfa for the concatenation of the
;;          languages of the given ndfas
;; Assume: The intersection of the states of the given
;;         machines is empty
(define (concat-fsa M N)
  (let* [(new-start (sm-start M))
         (new-sigma (remove-duplicates (append (sm-sigma M)
                                               (sm-sigma N))))
         (new-final (generate-symbol 'F (list 'F)))
         (new-states (cons new-final (append (sm-states M)
                                             (sm-states N))))
         (new-finals (list new-final)
         (new-rules (append (sm-rules M)
                            (sm-rules N)
                            (map (λ (f)
                                   (list f EMP (sm-start N)))
                                 (sm-finals M))))
                            (map (λ (f)
                                 (list f EMP new-final))
                                 (sm-finals N))))]
    (make-ndfa new-states
               new-sigma
               new-start
               new-finals
               new-rules)))
```

Is the famous professor correct? Prove or disprove the famous professor.

Fig. 37 Construction algorithm for L = L(M)*



8 Define and implement a concatenation constructor with the following signature:

```
ndfa ndfa → dfa
```

30.3 Kleene Star

Let the following be the machine that decides the language to be Kleene starred:

```
M = (make-ndfa S Σ A F δ)
```

We need to construct an `ndfa` that decides `L = L(M)*`. Recall that this means that `L` contains all words formed by concatenating zero or more words in `L(M)`.

The construction algorithm may be visualized as displayed in Fig. 37. Given that ϵ must be accepted by the constructed machine, a new state, `Z`, is generated that is both the start state and a final state. To the transitions, rules of `M` ϵ-transitions are added from `Z` to `S` and from every state in `F` to `Z`. The ϵ-transitions from the final states to `Z` implement loops that allow for another word in `L(M)` to be concatenated.

30.3.1 Implementation

As outlined by the design idea, to construct an `ndfa` for `L(M)*`, a new start state is generated, and the given `ndfa`'s alphabet is used. The set of states and the set of final states are obtained by adding the new start state, respectively, to the given `ndfa`'s states and final states. Finally, the set of rules is obtained by adding to the given `ndfa`'s rules the ϵ-transition from the new

Fig. 38 The Kleene star constructor implementation

```
;; ndfa → ndfa
;; Purpose: Construct ndfa for the Kleene star of given ndfa's language
(define (kstar-fsa M)
  (let* [(new-start (generate-symbol 'K (sm-states M)))
         (new-sigma (sm-sigma M))
         (new-states (cons new-start (sm-states M)))
         (new-finals (cons new-start (sm-finals M)))
         (new-rules (cons (list new-start EMP (sm-start M))
                          (append (sm-rules M)
                                  (map (λ (f) (list f EMP new-start))
                                       (sm-finals M)))))]
    (make-ndfa new-states new-sigma new-start new-finals new-rules)))

;; Tests for kstar-fsa
(define a-aUb-b*-* (kstar-fsa a-aUb-b*))
(define ab*-* (kstar-fsa ab*))

(check-equal? (sm-apply a-aUb-b*-* '(b b b)) 'reject)
(check-equal? (sm-apply a-aUb-b*-* '(a b a b a a a a)) 'reject)
(check-equal? (sm-apply a-aUb-b*-* '()) 'accept)
(check-equal? (sm-apply a-aUb-b*-* '(a a a a b b b b)) 'accept)
(check-equal? (sm-apply a-aUb-b*-* '(a a b a a b b a a)) 'accept)
(check-equal? (sm-testequiv? a-aUb-b*-* (sm-kleenestar a-aUb-b*)) #t)

(check-equal? (sm-apply ab*-* '(b)) 'reject)
(check-equal? (sm-apply ab*-* '(b b b)) 'reject)
(check-equal? (sm-apply ab*-* '()) 'accept)
(check-equal? (sm-apply ab*-* '(a a a a)) 'accept)
(check-equal? (sm-apply ab*-* '(a b a b b a b b b)) 'accept)
(check-equal? (sm-testequiv? ab*-* (sm-kleenestar ab*)) #t)
```

start state to the given machine's start state and the ϵ-transitions from the given `ndfa`'s final states to the new start state. These ϵ-transitions are computed by mapping a function to the given `ndfa`'s final states. The mapped function creates an `ndfa` rule using the given state, `EMP`, and the new start state. The implementation of this design idea is displayed in Fig. 38.

Tests are written using two machines created with the constructor. The tests illustrate the expected behavior of the constructed machine with concrete words and test the equivalence of the constructed machines with the machines obtained from using `FSM`'s `sm-kleenestar` constructor.

30.3.2 Algorithm Correctness Proof

Define two machines as follows:

`M = (make-ndfa S` Σ `Z F` δ`)`
`U = (kstar-fsa M) = (make-ndfa S'` Σ' `Z' F'` δ'`)`

Let `L = L(M)*`. We proceed to prove that `L = L(U)`.

Lemma 5 *w∈L ⇔ w∈L(U)*

Proof

(⇒) We need to show that w∈L ⇒ w∈L(U).

Assume w∈L. This means that $w = w_1w_2 \ldots w_n$, where $w_i \in$L(M). By construction of U, the following is a computation on w:

$((w_1w_2 \ldots w_n\ S') \vdash^*_U ((w_2 \ldots w_n)\ Y_1) \vdash^*_U ((\ldots w_n)\ Y_2) \vdash^*_U (()\ Y_n)$, where $Y_i \in$F'

Therefore, w∈L(U).

(⇐) We need to show that w∈L(U) ⇒ w∈L.

Assume w∈L(U). This means that $w = w_1w_2 \ldots w_n$ such that:

$((w_1 \ldots w_n)\ S') \vdash^*_U ((w_2 \ldots w_n)\ Y_1) \vdash^*_U ((w_3 \ldots w_n)\ Y_2) \ldots ((w_n)\ Y_{n-1}) \vdash^*_U (()\ Y_n)$, where $Y_i \in$F'.

By construction of U, F' contains S' and F. This means that w is the concatenation of zero or more words in L(M). Thus, w∈L. □

Lemma 6 *w∉L ⇔ w∉L(U)*

Proof

(⇒) We need to show that w∉L ⇒ w∉L(U).

Assume w∉L. This means that $w \neq w_1w_2 \ldots w_n$, where $w_i \in$L(M). By construction of U, the processing of w can occur in two ways. The first, w is consumed and U does not end in a final state. The second, w cannot be completely consumed. In both cases, w is rejected. Thus, w∉L(U).

(⇐) We need to show that w∉L(U) ⇒ w∉L.

Assume w∉L(U). This means that $w \neq w_1w_2 \ldots w_n$ such that:

$((w_1 \ldots w_n)\ S') \vdash^*_U ((w_2 \ldots w_n)\ Y_1) \vdash^*_U ((w_3 \ldots w_n)\ Y_2) \ldots ((w_n)\ Y_{n-1}) \vdash^*_U (()\ Y_n)$, where $Y_n \in$F'.

By construction of U, F' contains S' and F. This means that w is not the concatenation of zero or more words in L(M). Thus, w∉L. □

Theorem 4 *The languages accepted by finite-state machines are closed under Kleene star.*

Proof The theorem is established by Lemmas 5 and 6. □

9 A computer whiz claims to have a better implementation for `kstar-fsa`:

```
;; ndfa → ndfa
;; Purpose: Construct ndfa for the Kleene star of given
;;          ndfa's language
(define (kstar-fsa M)
  (let* [(new-start  (sm-start M))
         (new-sigma  (sm-sigma M))
         (new-states (sm-states M))
         (new-finals (if (member new-start (sm-finals M))
                         (sm-finals M)
                         (cons new-start (sm-finals M))))
         (new-rules (append (sm-rules M)
                    (map (λ (f) (list f EMP new-start))
                         (sm-finals M))))]
    (make-ndfa new-states
               new-sigma
               new-start
               new-finals
               new-rules)))
```

Is the computer whiz correct? Prove or disprove the claim made.

10 An up and coming computer science Ph.D. student claims that for `kstar-fsa`, the ϵ-transitions out of the final states can move the machine to the starting state of the given machine instead of the new starting state generated. Is she correct? Redesign `kstar-fsa` to use this new design, and prove that the new construction algorithm is correct.

11 Define and implement a Kleene star constructor with the following signature:

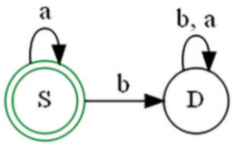
```
ndfa → dfa
```

30.4 Complement

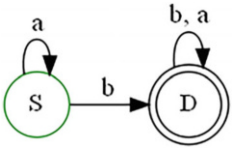
Let `M` be a `dfa`. The complement of `L(M)` is defined as follows:

$$\bar{L}(M) = \{w \mid w \notin L(M)\}$$

That is, the complement of `L(M)` is the language that contains all words not in `L(M)`. How can a machine for $\bar{L}(M)$ be constructed? The constructed machine must accept when `M` rejects and must reject when `M` accepts. Given that `M` is a `dfa`, this suggests inverting the roles of `M`'s states. That is, `M`'s final states are not final states in the constructed machine, and all other states in `M` are final states. For instance, consider the transition diagram for the `dfa` whose language is `L = a`*:



Reversing the roles of the states yields:



Indeed, reversing the roles of the states produces a `dfa` that accepts all string not in `L`.

To construct the complement `dfa`, only new final states need to be computed. Everything else remains unchanged. The new final states are all the states that are not final in the given `dfa`.

30.4.1 Implementation

The implementation of the complement constructor is displayed in Fig. 39. To compute the new final states, the states of the given `dfa` are filtered. The states that are not final states are kept as final states for the complement `dfa`. All other components of the given `dfa` are directly used to construct the complement `dfa`.

To test the complement constructor, two machines are constructed. The first is constructed for the sample `dfa a*` defined at the beginning of this chapter. The second is the `dfa` for the language of words containing an even number of `a`s and an odd number of `b`s defined in Sect. 24.5. For both, the tests illustrate the expected behavior of the complement machine with

Fig. 39 The complement constructor implementation

```
;; dfa → dfa
;; Purpose: Construct a dfa for the complement of given dfa's language
(define (complement-fsa M)
  (let* [(new-finals (filter (λ (s) (not (member s (sm-finals M))))
                             (sm-states M)))]
    (make-dfa (sm-states M)
              (sm-sigma M)
              (sm-start M)
              new-finals
              (sm-rules M)
              'no-dead)))

;; Tests for complement-fsa
(define not-a* (complement-fsa a*))
(define not-EVEN-A-ODD-B (complement-fsa EVEN-A-ODD-B))

(check-equal? (sm-apply not-a* '()) 'reject)
(check-equal? (sm-apply not-a* '(a a a)) 'reject)
(check-equal? (sm-apply not-a* '(a a b)) 'accept)
(check-equal? (sm-apply not-a* '(b b a a b)) 'accept)
(check-equal? (sm-testequiv? not-a* (sm-complement a*)) #t)

(check-equal? (sm-apply not-EVEN-A-ODD-B '(b)) 'reject)
(check-equal? (sm-apply not-EVEN-A-ODD-B '(a a b)) 'reject)
(check-equal? (sm-apply not-EVEN-A-ODD-B '(b b a b a)) 'reject)
(check-equal? (sm-apply not-EVEN-A-ODD-B '()) 'accept)
(check-equal? (sm-apply not-EVEN-A-ODD-B '(b b a a)) 'accept)
(check-equal? (sm-apply not-EVEN-A-ODD-B '(a a b b a b)) 'accept)
(check-equal? (sm-testequiv? not-EVEN-A-ODD-B
                             (sm-complement EVEN-A-ODD-B))
              #t)
```

concrete words and test for equivalence with the machine obtained using FSM's `(sm-complement)`.

30.4.2 Algorithm Correctness Proof

Define two machines as follows:

`M = (make-dfa S` Σ `Z F` δ`)`
`U = (complement-fsa M) = (make-ndfa S` Σ `Z F'` δ`)`

Let `L` $= \bar{L}$`(M)`. We proceed to prove that `L = L(U)`.

Lemma 7 *w∈L ⇔ w∉L(U)*

Proof

(⇒) We need to show that w∈L ⇒ w∉L(U).

Assume w∈L. Given that M is a `dfa`, the following is the computation performed on w:

((w) S) $\vdash^*_M$ (() Q), where Q∈F.

By construction of U, Q∉F', and M performs the same computation moving from S to Q by consuming w. Therefore, w∉L(U).

(⇐) We need to show that w∉L(U) ⇒ w∈L.

Assume w∉L(U). This means that U performs the following computation on w:

((w) S) $\vdash^*_U$ (() Q), where Q∉F'.

By construction of U, Q∈F, and M performs the same computation moving from S to Q by consuming w. Thus, w∈L. □

Lemma 8 *w∉L ⇔ w∈L(U)*

Proof

(⇒) We need to show that w∉L ⇒ w∈L(U).

Assume w∉L. Given that M is a `dfa`, the following is the computation it performs on w:

((w) S) $\vdash^*_M$ (() Q), where Q∉F.

By construction of U, Q∈F', and U performs the same computation moving from S to Q by consuming w. Therefore, w∈L(U).

(⇐) We need to show that w∈L(U) ⇒ w∉L.

Assume w∈L(U). This means that U performs the following computation on w:

((w) S) $\vdash^*_U$ (() Q), where Q∈F'.

By construction of U, Q∉F, and M performs the same computation moving from S to Q by consuming w. Thus, w∉L. □

Theorem 5 *The languages accepted by finite-state machines are closed under complement.*

Proof The theorem is established by Lemmas 7 and 8. □

12 The signature for `complement-fsa` requires as input a `dfa`. Why? What may go wrong if the input is an `ndfa`? Justify your answer.

13 In `complement-fsa`'s body, the optional argument `'no-dead` is provided to `make-dfa`. Why is this a good idea?

14 Computationally speaking, is `(complement-fsa (complement-fsa M)) = M`? Justify your answer.

30.5 Intersection

Let the following be the two machines that decide the languages whose intersection is needed:

```
M = (make-ndfa S_M Σ_M A F_M δ_M)
N = (make-ndfa S_N Σ_N R F_N δ_N)
```

We need to construct an `ndfa` that accepts and only accepts the words in `L(M)∩L(N)`, that is, an `ndfa` that only accepts the words that are both in `L(M)` and in `L(N)`. How is the intersection of two sets computed? Let Σ be an alphabet. Consider the following facts from set theory:

$$\Sigma^* - B = \{w \mid w \in \Sigma^* \wedge w \notin B\}$$
$$\Sigma^* - A = \{w \mid w \in \Sigma^* \wedge w \notin A\}$$

The first defines all possible words in Σ^* that are not in `B`. The second defines all possible words in Σ^* that are not in `A`. Consider the union of these two sets:

$$\{\Sigma^* - B\} \cup \{\Sigma^* - A\} = \{w \mid w \notin A \wedge w \notin B\}$$

What words are not contained in this union? It is exactly the elements that are in both `A` and `B`. Thus, we may define the language for the machine we wish to implement as follows:

$$\begin{aligned} L(M) \cap L(N) &= \Sigma^* - \{\{\Sigma^* - L(M)\} \cup \{\Sigma^* - L(N)\}\} \\ &= \Sigma^* - \{\bar{L}(M) \cup \bar{L}(N)\} \end{aligned}$$

Fig. 40 The intersection constructor implementation

```
;; ndfa ndfa → ndfa
;; Purpose: Construct an ndfa for the intersection of the languages of the
;;          given ndfas
(define (intersect-fsa M N)
  (let* [(notM (sm-rename-states (list DEAD) (sm-complement (ndfa->dfa M))))
         (notN (sm-rename-states (list DEAD) (sm-complement (ndfa->dfa N))))]
    (complement-fsa (ndfa->dfa (sm-union notM notN)))))

;; Tests for intersect-fsa
(define ab*-intersect-a-aUb-b* (intersect-fsa ab* a-aUb-b*))
(define a-aUb-b*-intersect-EVEN-A-ODD-B (intersect-fsa  a-aUb-b*
                                                        EVEN-A-ODD-B))

(check-equal? (sm-apply ab*-intersect-a-aUb-b* '()) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b b a)) 'reject)
(check-equal? (sm-apply ab*-intersect-a-aUb-b* '(a b)) 'reject)
(check-equal? (sm-testequiv? ab*-intersect-a-aUb-b*
                             (sm-intersection ab* a-aUb-b*))
              #t)

(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '()) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(b b)) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(a a b b)) 'reject)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(a a b)) 'accept)
(check-equal? (sm-apply a-aUb-b*-intersect-EVEN-A-ODD-B '(a a b b b)) 'accept)
(check-equal? (sm-testequiv? a-aUb-b*-intersect-EVEN-A-ODD-B
                             (sm-intersection a-aUb-b* EVEN-A-ODD-B))
              #t)
```

This means that the intersection machine is a machine for the complement of a machine for the union of the complement of `M` and the complement of `N`.

30.5.1 Implementation

We have a theoretical result that suggests how to implement the constructor:

$$\texttt{L(M)} \cap \texttt{L(N)} = \Sigma^* - \{\bar{\texttt{L}}\texttt{(M)} \cup \bar{\texttt{L}}\texttt{(N)}\}$$

According to the above equation, it suffices to complement M and N, union the two complement machines, and, finally, complement the obtained union machine. The following expression captures this idea:

```
(sm-complement (sm-union (sm-complement M)
                         (sm-complement N)))
```

Should this expression be packaged as the body of the intersection constructor? Answering this question requires carefully considering the `API` that is used. The first issue that arises is that the complement operation requires that the given machine be a `dfa`. `M` and `N` may be `ndfa`s. This means that any machine that is an `ndfa` must be converted to a `dfa`. This means that the expression above must be refactored to:

```
(sm-complement (sm-union (sm-complement (ndfa->dfa M))
                         (sm-complement (ndfa->dfa N))))
```

Can this expression be the body of the needed constructor? To answer this question, we must think carefully about what `ndfa->dfa`, `sm-complement`, and `sm-union` assume and do. Recall that the process of converting an `ndfa` to a `dfa` may introduce `FSM`'s dead state, `DEAD`, as the state representing the empty super state. This means that both `M` and `N` may have `DEAD` as a state. This is not a problem for `sm-complement`. Is it a problem for `sm-union`? Recall that constructing an `ndfa` for union assumes that the intersection of the states of the given machines is empty and, therefore, to compute the union of the states, all that is required is to append the list of states for each given machine. This raises a problem with the expression above. The complement of `M` and the complement of `N` may have a state name, `DEAD`, in common. This means that after appending the states for union, the resulting list may have a repeated state. As you may already know, `FSM` will not allow you to build a state machine with a list of states that contains a repetition. This means that the expression above must be further refactored. How can we guarantee that two machines do not share a name state? Perhaps, the easiest way to achieve this is to use `FSM`'s `sm-rename-states` to rename all the states of a given machine. This function takes as input a list of states that cannot appear in the renamed machine and the machine to rename. In this case, we do not want `DEAD` to be a state in the given machine. We can, therefore, refactor the above expression to:

```
(sm-complement
  (sm-union
    (sm-rename-states (list DEAD)
                      (sm-complement (ndfa->dfa M)))
    (sm-rename-states (list DEAD)
                      (sm-complement (ndfa->dfa N)))))
```

In this form, a union machine may be constructed, and its complement may also be constructed.

The implementation of the intersection constructor is displayed in Fig. 40. To make the code more readable, the two complement machines are locally defined. To test the constructor, two intersect machines are constructed. The

second machine is constructed using `EVEN-A-ODD-B` defined in Sect. 24.5. Observe that for the first machine, there are no tests with concrete words that lead to an accept. This is because the intersection of the two languages is empty and, therefore, the constructed machine does not have final state reachable from its start state. Both intersection machines are tested for equality with the machine obtained using `FSM`'s `sm-intersection`.

30.5.2 Algorithm Correctness Proof

Define three machines as follows:

```
M = (make-ndfa S Σ Z F δ)
N = (make-ndfa S' Σ' Z' F' δ')
U = (intersect-fsa M N) = (make-ndfa S'' Σ'' Z'' F'' δ'')
```

Let `L` = `L(M)` ∩ `L(N)`. We proceed to prove that `L = L(U)`.

Lemma 9 *w∈L ⇔ w∈L(U)*

Proof

(⇒) We need to show that w∈L ⇒ w∈L(U).

Assume w∈L. This means that w∈L(M) and w∈L(N). Therefore, w∈(Σ^* - {$\bar{L}$(M) ∪ $\bar{L}$(N)}). By construction of U, w∈L(U).

(⇐) We need to show that w∈L(U) ⇒ w∈L.

Assume w∈L(U). By U's construction, w∈(Σ^* - {$\bar{L}$(M) ∪ $\bar{L}$(N)}). Therefore, w∈L(M) and w∈L(N). Thus, w∈L. □

Lemma 10 *w∉L ⇔ w∉L(U)*

Proof

(⇒) We need to show that w∉L ⇒ w∉L(U).

Assume w∉L. This means that w∉L(M) or w∉L(N). Therefore, w∉(Σ^* - {$\bar{L}$(M) ∪ $\bar{L}$(N)}). By construction of U, w∉L(U).

(⇐) We need to show that w∉L(U) ⇒ w∉L.

Assume w∉L(U). By U's construction, w∉(Σ^* - {$\bar{L}$(M) ∪ $\bar{L}$(N)}). Therefore, w∉L(M) or w∉L(N). Thus, w∉L. □

Theorem 6 *The languages decided by finite-state automatons are closed under intersection.*

Proof The theorem is established by Lemmas 9 and 10. □

15 An alternative design for `intersect-dfa` has the constructor consume two `dfa`, `M` and `N`, and return a `dfa`. The returned `dfa` simulates both given `dfa`s simultaneously. This is achieved by having the states of the returned `dfa` represent super states: `(A B)`, where `A`∈`(sm-states M)` and `B`∈`(sm-states N)`. In essence, the returned `dfa` transitions between states that represent super states. After consuming a word, if the machine ends in a state that represents two final states (one for each given machine), the machine accepts. This design is akin to the design chosen to transform a `ndfa` to a `dfa`. Design, implement, and prove correct an intersection constructor using this design idea.

16 Let `M = (make-dfa S` Σ `Z F` δ`)`. Consider the following language:

$$\texttt{PREFIX} = \{\texttt{x} \mid \texttt{w} \in \texttt{L(M)} \wedge \texttt{w=xa} \wedge \texttt{x} \in \Sigma^* \wedge \texttt{a} \in \{\Sigma \cup \{\epsilon\}\}\}$$

Design, implement, and prove correct a `PREFIX` constructor for an arbitrary `M`.

31 Equivalence of Finite-State Machines and Regular Expressions

Recall that a regular language is generated by a regular expression using empty, the members of the alphabet, concatenation, union, and Kleene star. Figure 1 gives us good reason to believe that finite-state machines decide regular languages. That is, they decide if a word is or is not in a regular language.

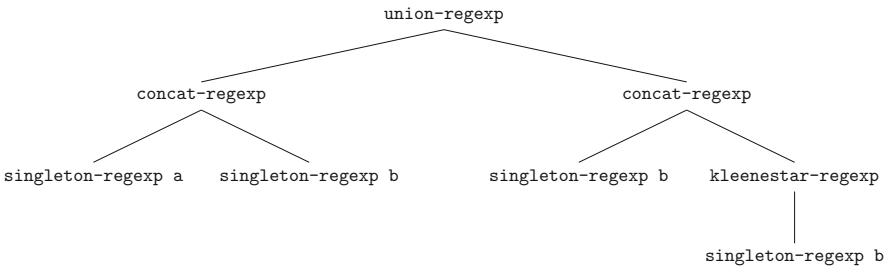
We may also ponder if every language that is decided by a finite-state machine is regular. That is, we may ask ourselves if there is a regular expression for the language decided by a finite-state machine.

Our goal in this section is to prove both of these, thus, establishing the equivalence of finite-state machines and regular expressions.

31.1 Creating an ndfa from a Regular Expression

We start by proving that there is a finite-state machine that decides the language of a regular expression. To prove this, a construction algorithm is presented to transform a regular expression into an ndfa.

Before proceeding to proving a theorem for this, it is useful to think about the structure of a regular expression. A regular expression may be thought of as a tree (specifically, as a binary tree). The empty regular expression and the singleton regular expressions are leaves. That is, they have no subtrees (i.e., contain no regular expressions). The concatenation, union, and Kleene star regular expressions are interior nodes (i.e., contain one or two regular expressions). Consider, for example, the regular expression for ab ∪ bb*. It has the following binary tree structure:



This is important because it suggests that a regular expression may be processed using structural recursion. Therefore, a construction algorithm to build an ndfa from a regular expression may be designed using structural recursion on a binary tree.

31.1.1 Construction Algorithm

To build an ndfa, a regular expression, e, and the language's alphabet, Σ, are needed. The subtype of the given regular expression is determined, and the appropriate ndfa is constructed. If the subtype is an empty regular expression, then an ndfa that only accepts empty is returned. If the subtype is a singleton regular expression for x∈Σ, then an ndfa that only accepts x is returned. These are the base cases for the recursion.

Now, we move to the recursive cases. If the subtype is a union regular expression, then Theorem 2 is used. That is, ndfas for the contained regular expressions are recursively constructed, and a machine that unions them is constructed. If the subtype is a concatenation regular expression, then Theorem 3 is used. That is, ndfas for the contained regular expressions are recursively constructed, and a machine that concatenates them is constructed. Finally, if the subtype is a Kleene star regular expression, then Theorem 4 is used. That is, an ndfa for the contained regular expression is recursively constructed, and from it, a Kleene star machine is constructed.

Fig. 41 The constructor to build an `ndfa` from a regular expression

```
;; regexp alphabet → ndfa
;; Purpose: Build an ndfa for the given regexp
(define (regexp->ndfa e sigma)
  (let* [(simple-tbl (map (λ (a)
                            (let [(S (generate-symbol 'S '(S)))
                                  (A (generate-symbol 'A '(A)))]
                              (list a (make-ndfa (list S A)
                                                 sigma
                                                 S
                                                 (list A)
                                                 (list (list S a A))))))
                          (cons EMP sigma)))]
    (cond [(empty-regexp? e) (second (assoc EMP simple-tbl))]
          [(singleton-regexp? e)
           (second (assoc (string->symbol (singleton-regexp-a e))
                          simple-tbl))]
          [(concat-regexp? e)
           (concat-fsa (regexp->ndfa (concat-regexp-r1 e) sigma)
                       (regexp->ndfa (concat-regexp-r2 e) sigma))]
          [(union-regexp? e)
           (union-fsa (regexp->ndfa (union-regexp-r1 e) sigma)
                      (regexp->ndfa (union-regexp-r2 e) sigma))]
          [else (kstar-fsa (regexp->ndfa (kleenestar-regexp-r1 e) sigma))])))
```

31.1.2 Implementation

The constructor's implementation is displayed in Fig. 41. It takes as input a regular expression, `e`, and the alphabet, `sigma`, for the language generated by the given regular expression. Locally, a table for simple `ndfa`s is computed. A function that creates a table entry is mapped to `EMP` and the given alphabet. Each table entry associates the given symbol with an `ndfa` that only accepts the given symbol. Each `ndfa` has only two states: a starting state and a final state. Each state is randomly generated, and the machine constructed has a single transition from the starting state on the given symbol to the final state.

The function employs structural recursion and dispatches on the given regular expression's type. If the given regular expression is for empty or a singleton, then the `ndfa` associated with it is extracted from the computed table. If it is a concatenation regular expression, then `ndfa`s for the contained regular expressions are recursively created, and an `ndfa` is constructed using `concat-fsa` from Fig. 36. If it is a union regular expression, then `ndfa`s for the contained regular expressions are recursively created, and an `ndfa` is constructed using `union-fsa` from Fig. 34. If it is a Kleene star regular expression, then an `ndfa` for the contained regular expression is recursively created, and an `ndfa` is constructed using `ksstar-fsa` from Fig. 38.

Fig. 42 The tests for `regexp->ndfa`

```
;; Tests for reg-exp->ndfa
(define e (empty-regexp))
(define a (singleton-regexp "a"))
(define b (singleton-regexp "b"))
(define ab (concat-regexp a b))
(define aa (concat-regexp a a))
(define abUe (union-regexp ab e))
(define abUaa (union-regexp ab aa))
(define aa-* (kleenestar-regexp aa))
(define abUaa-* (kleenestar-regexp abUaa))

(define Me (regexp->ndfa e '(a b)))
(define Ma (regexp->ndfa a '(a b)))
(define Mb (regexp->ndfa b '(a b)))
(define Mab (regexp->ndfa ab '(a b)))
(define Maa (regexp->ndfa aa '(a b)))
(define MabUMe (regexp->ndfa abUe '(a b)))
(define MabUaa (regexp->ndfa abUaa '(a b)))
(define Maa-* (regexp->ndfa aa-* '(a b)))
(define MabUaa-* (regexp->ndfa abUaa-* '(a b)))

(check-equal? (sm-apply Me '(a)) 'reject)
(check-equal? (sm-apply Me '()) 'accept)
(check-equal? (sm-apply Ma '(b)) 'reject)
(check-equal? (sm-apply Ma '(a)) 'accept)
(check-equal? (sm-apply Mab '()) 'reject)
(check-equal? (sm-apply Mab '(a b)) 'accept)
(check-equal? (sm-apply Maa '(b a a)) 'reject)
(check-equal? (sm-apply Maa '(a a)) 'accept)
(check-equal? (sm-apply MabUMe '(a b a a)) 'reject)
(check-equal? (sm-apply MabUMe '(b b)) 'reject)
(check-equal? (sm-apply MabUMe '()) 'accept)
(check-equal? (sm-apply MabUMe '(a b)) 'accept)
(check-equal? (sm-apply MabUaa '(a b b b)) 'reject)
(check-equal? (sm-apply MabUaa '(b a b)) 'reject)
(check-equal? (sm-apply MabUaa '(a a)) 'accept)
(check-equal? (sm-apply MabUaa '(a b)) 'accept)
(check-equal? (sm-apply Maa-* '(a b)) 'reject)
(check-equal? (sm-apply Maa-* '(a a a)) 'reject)
(check-equal? (sm-apply Maa-* '(a a)) 'accept)
(check-equal? (sm-apply Maa-* '(a a a a a a)) 'accept)
(check-equal? (sm-apply MabUaa-* '(a b a)) 'reject)
(check-equal? (sm-apply MabUaa-* '(b b b b)) 'reject)
(check-equal? (sm-apply MabUaa-* '()) 'accept)
(check-equal? (sm-apply MabUaa-* '(a a a a a b)) 'accept)
```

To test the constructor, sample regular expressions are defined and converted to finite-state machines. Each test applies a constructed machine to a word and illustrates the expected behavior. The tests are displayed in Fig. 42.

Run the tests and make sure they all pass. Take time to appreciate the elegance of `regexp->ndfa`. It is concise and very readable. It builds on theorems previously proven which leads to delegating `ndfa` building to auxiliary functions and, thus, allowing us to focus this function to the transformation steps required.

31.1.3 Correctness Proof

To establish correctness, we shall argue the correctness of the implementation. That is, we shall prove that `regexp->ndfa` is correct. Given that `regexp->ndfa` uses structural recursion on a binary tree, the proof is by induction on the height of the binary tree. In addition, it builds on closure properties for the languages decided by finite-state automatons as outlined in Fig. 1.

Theorem 7 *L is a regular language* $\Rightarrow$ *L is decided by a finite-state machine.*

Proof

Assume L is regular. This means that there is a regular expression, R, that defines L. Let Σ be the alphabet for the language of R. We prove by induction on the height of R that `(regexp->ndfa R` Σ`)` builds an `ndfa` for L.

Base Case: h = 0

If h is zero, then R must be an empty or a singleton regular expression. If R is the empty regular expression, the `(regexp->ndfa R` Σ`)` returns an `ndfa` that only accepts `EMP`. If R is `(singleton-regexp a)`, then `(regexp->ndfa R` Σ`)` returns an `ndfa` that only accepts `a`, where a$\in\Sigma$. This establishes the base case.

Inductive Step:
Assume: `(regexp->ndfa R` Σ`)` returns an `ndfa` that decides L for h = k.
Show: `(regexp->ndfa R` Σ`)` returns an `ndfa` that decides L for h = k+1.

h $\geq$ 0 $\Rightarrow$ h+1 $\geq$ 1. This means that R is a union, a concatenation, or a Kleene star regular expression. We analyze each regular expression subtype independently:

`(union-regexp S T)`

`(regexp->ndfa R` Σ`)` returns

```
(union-fsa (regexp->ndfa (union-regexp-r1 e) sigma)
           (regexp->ndfa (union-regexp-r2 e) sigma))
```

Observe that `(union-regexp-r1 e)`'s and `(union-regexp-r2 e)`'s height is at most k. By inductive hypothesis, therefore, the recursive calls return `ndfa`s for the language of each. By Fig. 2, `union-fsa` returns an `ndfa` for the union of these two languages.

`(concat-regexp S T)`

`(regexp->ndfa R` Σ`)` returns

```
(concat-fsa (regexp->ndfa (concat-regexp-r1 e) sigma)
            (regexp->ndfa (concat-regexp-r2 e) sigma))
```

Observe that `(concat-regexp-r1 e)`'s and `(concat-regexp-r2 e)`'s height is at most k. By inductive hypothesis, therefore, the recursive calls return `ndfa`s for the language of each. By Fig. 3, `concat-fsa` returns an `ndfa` for the concatenation of these two languages.

`(kleenestar-regexp S)`

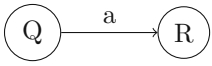
`(regexp->ndfa R` Σ`)` returns

```
(kstar-fsa (regexp->ndfa (kleenestar-regexp-r1 e) sigma))
```

Observe that `(kleenestar-regexp-r1 e)`'s height is at most k. By inductive hypothesis, therefore, the recursive call returns an `ndfa`, N, for its language. By Fig. 4, `kstar-fsa` returns an `ndfa` for the language containing the concatenation of an arbitrary number of words in L(N).

31.2 Creating a Regular Expression from an `ndfa`

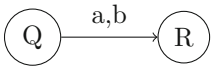
To create a regular expression for the language of an `ndfa`, a regular expression is needed that generates all words that take the given machine from its start state to any its final states. This means that a regular expression is needed from any state, `Q`, to any state, `R`, that is reachable from `Q` in the transition diagram of the machine. Consider, for example, the following portion of an `ndfa`:



`R` is reachable from `Q`, and, therefore, a regular expression is needed for the part of the word that takes the machine from `Q` to `R`. In this case, `Q` and `R` are neighbors, and a singleton regular expression is needed for the rule `(Q a R)`:

```
(singleton-regexp "a")
```

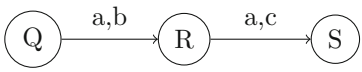
There can be more than one transition from `Q` to `R` as in the following machine snippet:



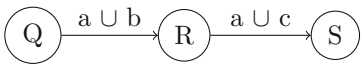
In this case, there are two transitions from `Q` to `R`: `(Q a R)` and `(Q b R)`. The machine can move from `Q` to `R` on a `a` or on a `b`. A regular expression, therefore, must be able to generate either an `a` or a `b`. The following is the needed regular expression:

```
(union-regexp (singleton-regexp "a")
              (singleton-regexp "b"))
```

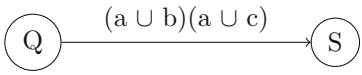
Now, consider the case when there are more than two connected nodes as in the following machine snippet:



There are four transitions: `(Q a R)`, `(Q b R)`, `(R a S)`, and `(R c S)`. Observe that `S` is reachable from `Q`. Therefore, a regular expression is needed to generate the part of the word that is consumed when the machine moves from `Q` to `S`. We already know how to deal with the transitions from `Q` to `R` and from `R` to `S`. Let us substitute the appropriate regular expressions in the diagram above:



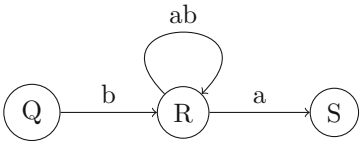
To get from `Q` to `S`, the word must contain an `a` or `b` concatenated with an `a` or `c`. The `R` state may be ripped out, along with the transitions into and out of it, and substituted with a transition from `Q` to `S` that concatenates the regular expression into `R` and the regular expression out of `R`. The result is visualized as follows:



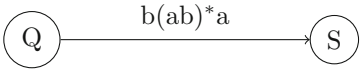
That is, the needed regular expression for the part of the word that takes the machine from `Q` to `R` is:

```
(concat-regexp (union-regexp (singleton-regexp a)
                             (singleton-regexp b))
               (union-regexp (singleton-regexp a)
                             (singleton-regexp c))
```

The intermediate node R may have an loop on it. That is, there is a transition from R to R as in the following machine snippet:



In this case, to rip out the intermediate state, a regular expression is needed that can generate the part of the word that takes the machine from Q to R, then generates zero or more times the part of the word that takes the machine from R to R, and finally generates the part of the word that takes the machine from R to S. Clearly, to generate a part of a word zero or more times, a Kleene star regular expression is needed. The result of ripping out R may be visualized as follows:



That is, the needed regular expression is:

```
(concat-regexp
  (singleton-regexp "b")
  (concat-regexp
    (kleenestar-regexp
      (concat-regexp (singleton-regexp "a")
                     (singleton-regexp "b")))
    (singleton-regexp "a")))
```

The above analysis suggests an algorithm to transform an ndfa into a regular expression. Given an ndfa, the goal is to construct a regular expression for all transition diagram paths from the start state to all reachable final states. To do so, a directed graph is created, and all the given machine's states are ripped out. The initial directed graph has all the machine's states as nodes, and the edges are labeled with a regular expression for what is consumed by a transition between two states. In addition, the initial directed graph has two extra nodes representing a new start state and a new and only final state. There is an ϵ-transition from the new start state to the machine's start state, and there are ϵ-transitions from every machine final state to the new final state. The process starts by collapsing multiple edges between two nodes into one labeled with a union regular expression. At each step, this graph is collapsed by ripping out a node representing a machine state. Ripping out a state may result in multiple edges between nodes, and these are collapsed before moving on to the next node to rip out. After all machine states are ripped out, the graph has been collapsed to two states (the new

start state and the new final state) with a single edge between them. The label on that edge is the regular expression for the machine's language.

To rip out a state, `S`, the graph's edges are partitioned into four subsets:

`not-s-edges` The list of edges that are not into nor out of `S`

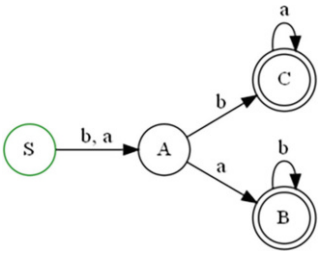
`into-s-edges` The list of non-loop edges that are into `S`

`outof-s-edges` The list of non-loop edges that are out of `S`

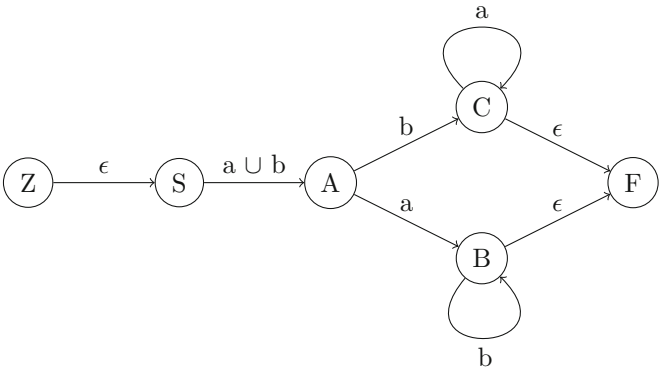
`self-edges` The list of self-loop edges on `S`

A new graph is constructed using `not-s-edges`, and the new edges are created using the other three sets of edges. If `S` has a self-loop, new edges are created for each incoming edge using the outgoing edges. Each new edge is from the start node of the incoming node to the destination node of an outgoing edge. The edge's label is a concatenation regular expression for the label of the incoming edge, the Kleene star of the self-loop label, and the label of an outgoing edge. If `S` does not have a self-loop, new edges are also created for each incoming edge using the outgoing edges. Each new edge is from the start node of the incoming node to the destination node of an outgoing edge. The edge's label is a concatenation regular expression for the label of the incoming edge and the label of an outgoing edge.

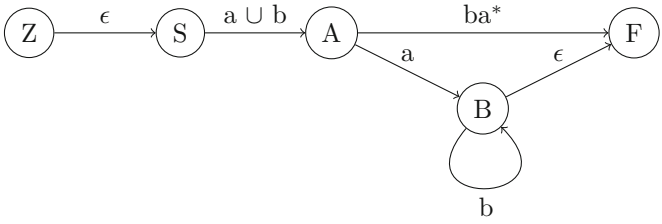
To illustrate the construction algorithm, consider transforming the following `ndfa`:



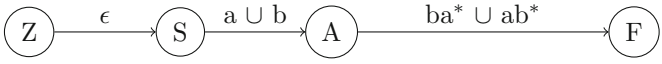
The initial graph constructed is:



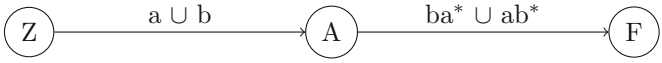
The edges between `S` and `A` are substituted with an edge with a union regular expression. To collapse the graph, at each step, a node representing a machine state is ripped out. The order in which they are ripped out does not matter. Let us start by ripping out `C`. `C` has a self-loop, the only into edge is `(A b C)`, and the only outgoing edge is `(C ϵ F)`. Ripping out `C`, therefore, results in a single new edge from `A` to `F` with `ba`* as its label.[9] The resulting graph is:



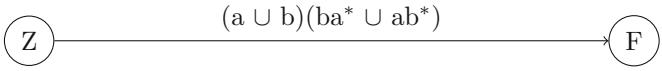
Let us now rip out `B`. To remove this node, the edges `(A a B)`, `(B b B)`, and `(B ϵ F)` need to be collapsed. Collapsing these results in two edges, `(A ba`* `F)` and `(A ab`* `F)`, from `A` to `F` that must be collapsed into one. The resulting graph is:



Next, let us rip out `S`. The edges `(Z ϵ S)` and `(S a∪b A)` are collapsed into an edge from `Z` to `A`. The resulting graph is:



The only node left to be ripped out is `A`. It does not have a self-loop and a single incoming edge from `Z` and a single outgoing edge to `F`. Therefore, these edges are collapsed into an edge from `Z` to `F` by concatenating their labels. The resulting graph is:



The label on the only remaining edge represents the regular expression for the language of the given machine. The regular expression is:

[9] $ba^*\epsilon = ba^*$.

```
(concat-regexp
  (union-regexp (singleton-regexp "a")
                (singleton-regexp "b"))
  (union-regexp
    (concat-regexp
      (singleton-regexp "b")
      (kleenestar-regexp (singleton-regexp "a")))
    (concat-regexp
      (singleton-regexp "a")
      (kleenestar-regexp (singleton-regexp "b")))))
```

The discussion so far has assumed that the language of the given machine, `M`, is not empty. That is, `L(M)` $\neq \varnothing$. If the language of the given machine is empty, then the collapsed graph will have no edges. This is a problem because there is no regular expression for generating no words. To address this problem, `FSM` introduces a new regular expression constructor, `null-regexp`, to represent a language with no words.

31.2.1 Implementation

The transformation from a `ndfa` to a regular expression requires the creation of a directed graph. A directed graph is defined and documented as follows:

```
;; Data Definitions
;;
;; A node is a symbol
;;
;; An edge, (list node regexp node), has a beginning
;; node, a regular expression for its label, and
;; destination node.
;;
;; A directed graph, dgraph, is a (listof edge)
```

That is, a directed graph is a list of edges, and an edge is a list with three elements: a node, a regular expression, and a node, where node is a symbol. To test the constructor, `EVEN-A-ODD-B` from Sect. 24.5 and the following machines are used:

Fig. 43 The constructor for a regular expression from an `ndfa`

```
;; ndfa → regexp
;; Purpose: Create a regexp from the given ndfa
;; Assume: The transition diagram of the given machine is a connected
;;         directed graph
(define (ndfa2regexp m)
  (let* [(new-start (generate-symbol 'S (sm-states m)))
         (new-final (generate-symbol 'F (sm-states m)))
         (init-dgraph (make-dgraph
                       (cons (list new-start EMP (sm-start m))
                             (append (map (λ (f) (list f EMP new-final))
                                          (sm-finals m))
                                     (sm-rules m)))))
         (collapsed-dgraph
          (rip-out-nodes (sm-states m) (remove-multiple-edges init-dgraph)))]
    (if (empty? collapsed-dgraph)
        (null-regexp)
        (simplify-regexp (second (first collapsed-dgraph))))))

;; Tests for ndfa2regexp
(check-equal? (printable-regexp (ndfa2regexp EMPTY))
              "()")
(check-equal? (printable-regexp (ndfa2regexp b*))
              "b*")
(check-equal? (printable-regexp (ndfa2regexp ab*Uaa*))
              "(b U a)(ab* U ba*)")
(check-equal?
 (printable-regexp (ndfa2regexp EVEN-A-ODD-B))
 "((ba U ab)(aa U bb)*(ab U ba) U (aa U bb))*((ba U ab)(aa U bb)*a U b)")
```

```
;; L =
(define EMPTY (make-ndfa '(S) '(a b) 'S '() '()))

;; L = ab* U ba*
(define aUb-ba*Uab*
        (make-ndfa
          '(S A B C)
          '(a b)
          'S
          '(B C)
          '((S a A) (S b A) (A a B) (A b C) (B b B) (C a C))))

;; L = b*
(define b* (make-ndfa `(,DEAD S A)
                      '(a b)
                      'S
                      '(A)
                      `((S ,EMP A) (S a ,DEAD) (A b A))))
```

Fig. 44 The constructor for the initial directed graph

```
;; (listof ndfa-rule) → dgraph
;; Purpose: Create a dgraph from the given ndfa
(define (make-dgraph lor)
  (map (λ (r) (if (eq? (second r) EMP)
                  (list (first r) (empty-regexp) (third r))
                  (list (first r)
                        (singleton-regexp (symbol->string (second r)))
                        (third r))))
       lor))

;; Tests for make-dgraph
(check-equal? (make-dgraph '()) '())

(check-equal?
 (make-dgraph `((S ,EMP A) (S a ,DEAD) (A b A)))
 (list (list 'S (empty-regexp) 'A)
       (list 'S (singleton-regexp "a") 'ds)
       (list 'A (singleton-regexp "b") 'A)))

(check-equal?
 (make-dgraph '((S a A) (S b A) (A a B) (A b C) (B b B) (C a C)))
 (list (list 'S (singleton-regexp "a") 'A)
       (list 'S (singleton-regexp "b") 'A)
       (list 'A (singleton-regexp "a") 'B)
       (list 'A (singleton-regexp "b") 'C)
       (list 'B (singleton-regexp "b") 'B)
       (list 'C (singleton-regexp "a") 'C)))
```

The constructor takes as input an `ndfa`, `m`, and returns a regular expression. It generates and locally defines the names for two new states. It creates and locally defines the initial directed graph from `m`'s rules, the new ϵ-transition from the new starting state to `m`'s starting state, and the new ϵ-transitions from `m`'s final states to the new final state. It then creates and locally defines the collapsed graph by collapsing multiple edges between states and ripping out `m`'s state from the initial graph. The collapsed graph is tested to determine if it is empty. If so, a null regular expression is returned because `m`'s language is empty. Otherwise, the regular expression in its only edge is simplified and returned. The implementation of this design is displayed in Fig. 43. The tests use the testing machines mentioned above. To make each test more readable, a machine is transformed, and the resulting regular expression is converted to a string that is tested. All but the last test are fairly straightforward to visually verify as producing the right regular expression. For the last test, observe that the first part of the regular expression (up to and including the second Kleene star) produces an even number of `a`s and an even number of `b`s. Such a partial word is concatenated with the result of the rest of the regular expression that always produces an even number of `a`s and an odd number of `b`s. Thus, it produces a word with an even number of `a`s and an odd number of `b`s.

Fig. 45 The function to collapse multiple edges between nodes

```
;; dgraph → dgraph
;; Purpose: Collapse multiple edges between nodes
;; Accumulator Invariant: g = the unprocessed graph
(define (remove-multiple-edges g)
  (if (empty? g)
      '()
      (let* [(curr-edge (first g))
             (from-state (first curr-edge))
             (to-state (third curr-edge))
             (to-collapse (filter (λ (e) (and (eq? (first e) from-state)
                                              (eq? (third e) to-state)))
                                  g))
             (remaining-g (filter (λ (e) (not (member e to-collapse))) g))]
        (cons (list from-state (collapse-edges to-collapse) to-state)
              (remove-multiple-edges remaining-g)))))

;; Tests for remove-multiple-edges
(check-equal? '() '())
(check-equal?
 (remove-multiple-edges `((S ,(singleton-regexp "a") A)
                          (S ,(singleton-regexp "b") A)
                          (A ,(singleton-regexp "a") A)))
 `((S
    ,(union-regexp (singleton-regexp "a") (singleton-regexp "b"))
    A)
   (A ,(singleton-regexp "a") A)))
```

The constructor for the initial `dgraph` takes as input a list of `ndfa` rules and returns a `dgraph`. It maps a function onto the given list of rules. This mapped function produces an edge for the given rule. It examines the consumed element in the given rule to decide if the edge's regular expression ought to be an empty or a singleton regular expression. The result of this design is displayed in Fig. 44.

The function `remove-multiple-edges` collapses multiple edges between nodes. It takes as input a `dgraph` and returns a `dgraph`. The input is an accumulator for an unprocessed graph. It divides the edges in the given `dgraph` in two: those that are between the same nodes as the given `dgraph`'s first edge and those that are not. A new graph is created from a collapsed new edge for the first set of edges and recursively processing the graph that only contains the second set of edges. The result of this design is displayed in Fig. 45. The function to collapse edges takes as input a list of edges that are all between the same nodes and returns a union regular expression. The function recursively process the given list of edges and may be implemented as follows:

Fig. 46 The function to rip out an arbitrary number of states

```
;; (listof node) dgraph → dgraph
;; Purpose: Rip out the given nodes from the given graph
;; Assume: Given nodes in given graph and g has no multiple edges
;;         between nodes
(define (rip-out-nodes lon g)
  (foldr (λ (s g) (rip-out-node s g)) g lon))

;; Tests for rip-out-nodes
(check-equal? (rip-out-nodes '() `((S ,(singleton-regexp "a") A)
                                   (A ,(singleton-regexp "b") B)))
              `((S ,(singleton-regexp "a") A)
                (A ,(singleton-regexp "b") B)))

(check-equal?
  (rip-out-nodes '(A B) `((S ,(singleton-regexp "a") A)
                          (A ,(singleton-regexp "b") B)
                          (B ,(singleton-regexp "b") C)))
  `((S
     ,(concat-regexp (singleton-regexp "a")
                     (concat-regexp (singleton-regexp "b")
                                    (singleton-regexp "b")))
     C)))
```

```
;; (listof edge) → regexp
;; Purpose: Collapse the given edges into a regexp
(define (collapse-edges loe)
  (cond [(empty? loe) '()]
        [(empty? (rest loe)) (second (first loe))]
        [else (union-regexp (second (first loe))
                            (collapse-edges (rest loe)))]))
;; Tests for collapse-edges
(check-equal? (collapse-edges '()) '())
(check-equal? (collapse-edges `((S ,(singleton-regexp "a") S)))
              (singleton-regexp "a"))
(check-equal?
 (collapse-edges `((A ,(singleton-regexp "a") A)
                   (A ,(singleton-regexp "b") A)
                   (A ,(empty-regexp) A)))
 (union-regexp (singleton-regexp "a")
               (union-regexp (singleton-regexp "b")
                             (empty-regexp))))
```

The function to rip out an arbitrary number of nodes from a graph assumes all the given nodes are in the given graph. It traverses the given list of nodes using `foldr`. The accumulator represents a graph. Initially, the accumulator is the given graph. For each node, a new a new graph is computed by ripping out the node from the accumulator. This graph becomes the new value of the accumulator to process the next node. The result of this design is displayed

Fig. 47 The function to rip out a node

```
;; node dgraph → dgraph
;; Purpose: Rip out given state from given graph
(define (rip-out-node n g)
  ;(define d (displayln (format "removing: ~s\n from: ~s\n" s g)))
  (let* [(non (filter (λ (r) (and (not (eq? (third r) n))
                                  (not (eq? (first r) n))))
                      g))
         (into-n (filter (λ (r) (and (eq? (third r) n)
                                     (not (eq? (first r) n))))
                         g))
         (outof-n (filter (λ (r) (and (eq? (first r) n)
                                      (not (eq? (third r) n))))
                          g))
         (self-edges (filter (λ (r) (and (eq? (first r) n)
                                         (eq? (third r) n)))
                             g))]
    (remove-multiple-edges
     (append
      non
      (if (not (empty? self-edges))
          (let [(self-edge (first self-edges))]
            (append-map
             (λ (into-edge)
               (map (λ (outof-edge)
                      (list (first into-edge)
                            (concat-regexp
                             (second into-edge)
                             (concat-regexp
                              (kleenestar-regexp (second self-edge))
                              (second outof-edge)))
                            (third outof-edge)))
                    outof-n))
             into-n))
          (append-map (λ (into-edge)
                        (map (λ (outof-edge)
                               (list (first into-edge)
                                     (concat-regexp (second into-edge)
                                                    (second outof-edge))
                                     (third outof-edge)))
                             outof-n))
                      into-n))))))
```

in Fig. 46. The first test illustrates that the given graph is returned when there are no nodes to remove. The second test illustrates the graph returned when there are nodes to remove.

The function to rip out a given node from a given graph assumes the given node is in the given graph. It partitions the edges in four: edges that do not contain the given node, edges into the given node that are not self-loops, edges out of the given node that are not self-loops, and edges that are a self-

Fig. 48 The tests for the function to rip out a node

```
;; Tests for rip-out-node
(check-equal?
 (rip-out-node
   'A
   `((S ,(singleton-regexp "a") A) (A ,(singleton-regexp "b") B)))
   `((S ,(concat-regexp (singleton-regexp "a") (singleton-regexp "b")) B)))
(check-equal?
  (rip-out-node
    'C
    `((S ,(singleton-regexp "a") A) (S ,(singleton-regexp "b") B)
      (A ,(singleton-regexp "a") C) (B ,(singleton-regexp "b") C)
      (C ,(singleton-regexp "a") D) (C ,(singleton-regexp "b") E)))
  `((S ,(singleton-regexp "a") A)
    (S ,(singleton-regexp "b") B)
    (A ,(concat-regexp (singleton-regexp "a") (singleton-regexp "a")) D)
    (A ,(concat-regexp (singleton-regexp "a") (singleton-regexp "b")) E)
    (B ,(concat-regexp (singleton-regexp "b") (singleton-regexp "a")) D)
    (B ,(concat-regexp (singleton-regexp "b") (singleton-regexp "b")) E)))
```

loop. The self-edges are examined to determine if they are empty or not. If there is a self-edge, then for each out-edge, `oe`, the into-edges are traversed. For each into-edge, `ie`, a new edge is created using `ie`'s starting node and `oe`'s destination node. Then the new edge's label is a concatenation regular expression for `ie`'s regular expression, a Kleene star regular expression for the self-loop's regular expression, and `oe`'s regular expression. If there is not a self-edge, then for each out-edge, `oe`, the into-edges are traversed. For each into-edge, `ie`, a new edge is created using `ie`'s starting node and `oe`'s destination node. Then the new edge's label is a concatenation regular expression for `ie`'s regular expression and `oe`'s regular expression. The implementation of this design is displayed in Fig. 47. The tests for this function are displayed in Fig. 48. The first test illustrates ripping out of a node with a single incoming edge and a single outgoing edge. The second test illustrates ripping out a node with multiple incoming and outgoing edges.

31.2.2 Correctness Proof

The proof of correctness requires proving that all functions return the expected value. To prove that each function is correct, assume that the auxiliary functions are correct. We start with the main function.

Theorem 8 *(ndfa2regexp m) returns a regular expression for L(m).*

Proof
New start and final states are generated. To m's rules, ϵ-transitions are added from the new start state to m's start state and from each of m's final states to the new final state. By assumption, `make-dgraph` creates the correct initial

directed graph. Multiple edges between any pair of nodes are removed by `remove-multiple-edges`, and all the nodes that represent a state in m are ripped out by `rip-out-nodes` to create the collapsed graph. Observe that the graph meets the assumptions made by `rip-out-nodes`. These auxiliary functions, by assuming their correctness, return the correct graph for their given input. The collapsed graph is examined to test if it is empty. If so, `(null-regexp)` is returned. This is correct because L(m) is empty. Otherwise, the only edge's regular expression is returned. This is correct because this regular expression can generate words on all paths from the new start state to the new final state in the initial directed graph. □

We proceed to argue the correctness of `rip-out-nodes`.

Theorem 9 *(rip-out-nodes lon g) returns a dgraph resulting from removing the given nodes from the given graph.*

Proof
By assumption, all the given nodes are in the given graph, and the given graph does not have multiple edges between nodes. The given list of nodes is traversed using `foldr` to rip out one node at a time from the given graph. Initially, `foldr`'s accumulator is the given graph. For each node, `foldr` creates a new graph by ripping out the next node using `rip-out-node`. Observe that the graph returned by `rip-out-node` satisfies the assumptions made by this function. Therefore, the returned graph may be input to `rip-out-node`. This auxiliary function, by assumption, is correct. Thus, `rip-out-nodes` returns the correct directed graph after ripping out all the given nodes. □

Next, argue for the correctness of `rip-out-node`.

Theorem 10 *(rip-out-node n g) returns a dgraph resulting from removing the given node from the given dgraph.*

Proof
By assumption, the given node is in the given graph. The given graph is filtered four times to extract four mutually exclusive sets of rules:

`non` The set of edges that are not into nor out of n. The graph is properly filtered given that only edges that do not have n as a starting node and do not have n as a destination node are extracted.

`into-n` The set of edges into n. The graph is properly filtered given that only edges that do not have n as a starting node and do have n as a destination node are extracted.

`outof-n` The set of edges out of n. The graph is properly filtered given that only edges that have n as a starting node and do not have n as a destination node are extracted.

`self-edges` The set of edges that are self-loops on n. The graph is properly filtered given that only edges that have n as a starting node and have n as a destination node are extracted.

Observe that there can only be at most one self-loop on n because by assumption, the graph does not have multiple edges between nodes. If there is a self-loop on n, then new edges are created using `into-n`, `self-edges`, and `outof-n`. For each edge, i, in `into-n`, a new edge is created using each edge, o, in `outof-n` that has the following form:

```
(starting state of i
 (concat-regexp
   regular expression in i
   (concat-regexp
     (kleenestar-regexp regular expression in only self-edge)
     regular expression in o))
 destination state of o)
```

This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the destination state of o. If there is no self-loop on n, then new edges are created using `into-n` and `outof-n`. For each edge, i, in `into-n`, a new edge is created using each edge, o, in `outof-n` that has the following form:

```
(starting state of i
 (concat-regexp
   regular expression in i
   regular expression in o)
 destination state of o)
```

This is correct because the regular expression can generate all words that take the machine from the state represented by the starting node of i to the state represented by the destination node of o. □

The remaining proofs for auxiliary functions are left as exercises.

17 Prove `make-dgraph`'s correctness.

18 Prove by induction on the length of the given list of edges `collapse-edges`' correctness.

19 Prove `remove-multiple-edges`'s correctness by induction on the number of calls to it. Recall that every time the function is called, the accumulator invariant must hold.

20 It is not always necessary to add a new start state and a new final state to the initial directed graph. A new start state is unnecessary if the given machine's start state is not the destination state for any transition. A new final state is unnecessary if the given machine only has one final state and this final state is not the start state for any transition. Refine the program to transform an `ndfa` to a regular expression taking these observations into account and prove its correctness.

Chapter 8
Regular Grammars

In this chapter, we explore grammars as a means to specify the generation of words in a language. Specifically, we explore regular grammars that specify the rules for generating words in a regular language. You are probably, to some extent, familiar with grammars from a past course in programming languages or in English grammar. In an English grammar class, for example, you may have been told that a sentence is a subject followed by a predicate or a sentence is a noun followed by a verb followed by a noun. These concepts may have been presented as production rules that look like this:

```
<sentence> → <subject> <predicate>
<sentence> → <noun> <verb> <noun>
```

The elements inside angled brackets represent syntactic categories. A syntactic category represents something that must be generated. For the production rules above, the syntactic categories are described as follows:

<sentence>: Generates a sentence

<subject>: Generates a subject

<predicate>: Generates a predicate

<noun>: Generates a noun

<verb>: Generates a verb

If we also have the following rules:

```
<noun> → humans
<noun> → cake
<verb> → eat
```


M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_8

The sentence *Humans eat cake* may be derived using the production rules above as follows:

```
<sentence> → <noun> <verb> <noun>
           → humans <verb> <noun>
           → humans eat <noun>
           → humans eat cake
```

At each step, a syntactic category on the left-hand side of a production rule is substituted with the right-hand side of the production rule. Using the same rules, the sentence *Cake eat humans* may also be generated – which is not proper English. Fortunately, we shall not start with languages as complex as the English language. We shall start with regular languages.

32 Regular Grammars

We know that a `dfa` decides a regular language by reading one symbol at a time. This suggests that words in the language may be generated one symbol at a time starting from, `S`, an initial syntactic category. We have to, of course, be able to generate the empty word because it may be part of a regular language. Based on these observations, we shall say that there are three types of rules in a regular grammar:

1. S generates the empty word (i.e., `EMP`).
2. Rules that generate an alphabet member.
3. Rules that generate a symbol representing the concatenation of a terminal symbol and a symbol representing a syntactic category.

Historically, the members of the alphabet have been called the terminal symbols, and the symbols representing syntactic categories are called the nonterminals.

We can now formally define a regular grammar as follows:

`A regular grammar is an instance of (make-rg N` Σ `R S)`

`N` is the set of capital letters in the Roman alphabet representing the nonterminal symbols (i.e., syntactic categories). Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (used to construct words). `S` is the starting nonterminal symbol. `R` is the set of generating (or production) rules. Each production rule contains a nonterminal followed by an arrow and a symbol. There are only three types of production rules:

`S` $\rightarrow \epsilon$`, where S is the starting nonterminal and` $S \in$ N
`A` $\rightarrow$ `a,  where` $A \in$ N `and` $a \in \Sigma$
`A` $\rightarrow$ `aB, where A,B`$\in$`N and a`$\in\Sigma$

Observe that each rule generates one terminal symbol at a time. The language of a grammar `G` is denoted as `L(G)`. It contains all the words that can be generated using `G`.

`FSM` provides a set of grammar observers for programmers:

`(grammar-nts g)`: Returns a list of `g`'s nonterminal symbols.
`(grammar-sigma g)`: Returns a list of `g`'s terminal symbols.
`(grammar-rules g)`: Returns a list of `g`'s production rules.
`(grammar-start g)`: Returns `g`'s starting nonterminal.
`(grammar-type g)`: Returns a symbol for `g`'s grammar type.
`(grammar-derive g w)`: If the given word, `w`, is in the language of the given grammar, then a derivation for `w` is returned. Otherwise, a string indicating that `w` is not in the language of the given grammar is returned.

A derivation consists of one or more derivation steps. A derivation step is the application of a production rule and is denoted by $\rightarrow$. One or more derivation steps is denoted by $\rightarrow^+$.

In addition, `FSM` provides the following testing functions for grammars:

`(grammar-both-derive g1 g2 w)`: Tests if both of the given grammars obtain the same result when trying to derive the given word.
`(grammar-testequiv g1 g2 [natnum])`: Tests if the given grammars obtain the same results when deriving 100 (or the optional number of) randomly generated words. If all tests give the same result, true is returned. Otherwise, a list or words that produce different results is returned.
`(grammar-test g1 [natnum])`: Tests the given grammar with 100 (or the optional number of) randomly generated words. A list of pairs containing a word and the result of attempting to derive the word are returned.

Just as finite-state machines are designed, grammars also need to be designed. For finite-state machines, the design revolves around the states needed to decide if a word is in the language. For regular grammars, the design revolves around the syntactic categories needed to generate a word. Each nonterminal represents a type (of partial) word that must be generated. Let us explore this by implementing a regular grammar for:

$\texttt{L} = \{\epsilon\} \cup \texttt{ba}^*$

We start by picking a descriptive name for the grammar and specifying its alphabet:

```
Name: EMP-U-ba*
Σ: (a b)
```

Next, we need to define the nonterminals (i.e., syntactic categories) needed. Usually, this process starts with the starting nonterminal. This nonterminal generates any word in the language. For `L`, we define the starting nonterminal as follows:

$\texttt{S represents } \epsilon \wedge \texttt{ a word in ba}^*$

If the word is not empty, then it may contain a `b` followed by an arbitrary number of `a`s. We can distinguish two cases here. The first is when no `a`s are generated. In this case, there is no need to generate a nonterminal to generate `a`s. The second is when one or more `a`s needs to be generated. An arbitrary number of one or more `a`s is a different type of word that must be generated and, therefore, requires a syntactic category:

```
A represents an arbitrary number of 1 or more as
```

No other types of words need to be generated. This means only the two nonterminals above are needed to build the grammar.

The next step is to develop the production rules. Given that the empty word is part of the language, the starting nonterminal must generate it. The needed production rule is:

```
(list 'S ARROW EMP)
```

`ARROW` is the `FSM` constant denoting an arrow in a production rule. The starting nonterminal must also generate words that start with a `b` followed by an arbitrary number of `a`s. This means that `S` ought to generate either a `b` (i.e., not generate any `a`s) or a `b` followed by `A` (the nonterminal to generate one or more `a`s). The needed production rules are:

```
(list 'S ARROW 'b)
(list 'S ARROW 'bA)
```

Finally, the production rules for `A` to generate one or more `a`s are needed. `A` ought to generate an `a` (i.e., only one `a`) or an `a` followed by one or more `a`s. Observe that the latter is `A`'s definition. Thus, the needed production rules are:

```
(list 'A ARROW 'a)
(list 'A ARROW 'aA)
```

Next, we define tests to illustrate words that are and that are not in the grammar's language. To do so, we need to be more specific about `grammar-derive`'s output. When the given word is not in the language, the returned string has this structure:

```
"<given word> is not in L(G)."
```

In the returned string, `<given word>` is substituted with the word given to `grammar-derive`. When the given word is in the language, its derivation is returned. A derivation is a list that starts with the starting nonterminal and ends with a symbol representing the given word. In between, there is a step for each rule applied. For example, the following is the derivation for `'(a b)`:

```
'(S -> bA -> ba)
```

The derivation starts with `S`, and the third rule is applied to substitute `S` to obtain `bA`. In the next step, the `A` is substituted using the fourth rule

Fig. 49 The regular grammar implementation for $\mathtt{L} = \{\epsilon\} \cup \mathtt{ba}^*$

```
#lang fsm

;; L = ϵ U ba*
(define EMP-U-ba* (make-rg '(S A)
                           '(a b)
                           `((S ,ARROW ,EMP)
                             (S ,ARROW b)
                             (S ,ARROW bA)
                             (A ,ARROW a)
                             (A ,ARROW aA))
                           'S))

;; Tests for EMP-U-ba*
(check-equal? (grammar-derive EMP-U-ba* '(a))
              "(a) is not in L(G).")
(check-equal? (grammar-derive EMP-U-ba* '(a b b a a))
              "(a b b a a) is not in L(G).")
(check-equal? (grammar-derive EMP-U-ba* '())
              '(S -> ϵ))
(check-equal? (grammar-derive EMP-U-ba* '(b a))
              '(S -> bA -> ba))
(check-equal? (grammar-derive EMP-U-ba* '(b a a a))
              '(S -> bA -> baA -> baaA -> baaa))
```

to obtain ba. There are no more nonterminals to substitute, and, therefore, the derivation is complete. With this understanding, tests may be written as follows:

```
;; Tests for EMP-U-ba*
(check-equal? (grammar-derive EMP-U-ba* '(a))
              "(a) is not in L(G).")
(check-equal? (grammar-derive EMP-U-ba* '(a b b a a))
              "(a b b a a) is not in L(G).")
(check-equal? (grammar-derive EMP-U-ba* '())
              '(S -> ϵ))
(check-equal? (grammar-derive EMP-U-ba* '(b a a a))
              '(S -> bA -> baA -> baaA -> baaa))
```

The program implementing the regular grammar designed is displayed in Fig. 49. Run the tests and make sure they pass.

Fig. 50 The design recipe for grammars

1. Pick a name for the grammar and specify the alphabet
2. Define each syntactic category and associate each with a nonterminal clearly specifying the starting nonterminal
3. Develop the production rules
4. Write unit tests
5. Implement the grammar
6. Run the tests and redesign if necessary

33 The Design Recipe for Grammars

Building on the experience implementing the grammar in Fig. 49, the steps of a design recipe for grammars are displayed in Fig. 50. Step 1 asks you to select a descriptive name for the grammar and specify the alphabet for the grammar's language.

Step 2 asks you to identify the syntactic categories that are needed and associate each with a nonterminal. This step usually starts with the starting nonterminal. The starting nonterminal represents all the words in the grammar's language. From there, follow a top-down approach identifying the different parts of a word that must be generated. For each part that must be generated, use a different nonterminal, or use a previously defined nonterminal.

Step 3 asks you to develop the production rules. This step is tightly coupled with step 2. Each production rule has a nonterminal on the left-hand side before the arrow and a symbol on the right-hand side after the arrow. A nonterminal may have more than one production rule. Collectively, the production rules for a nonterminal define what the nonterminal may be substituted by in one derivation step. A useful heuristic to follow is to separate the generation of an arbitrary number of elements into two parts: the generation of zero of those elements and the generation of one or more of those elements (as done for `a`* in the development of the regular grammar in Fig. 49).

Step 4 asks you to write unit tests to illustrate the expected derivations using the grammar. Write tests for words that are not in the grammar's language and tests for words that are in the grammar's language. Make sure that, collectively, the tested words use every production rule in the grammar.

Step 5 asks you to implement the grammar using a grammar constructor (e.g., `make-rg`) and the results of the previous steps.

Step 6 asks you to run the tests and redesign if necessary. If you provide the wrong type of arguments to the constructor, carefully read the error messages. The `FSM` error messages are designed to be informative. They do not, however, prescribe a solution because it is impossible to know the intentions of the programmer.

34 The Design Recipe in Action

Consider designing and implementing a regular grammar for:

L = {w | the number of as in w is a multiple of 3}

Let us work through the steps of the design recipe for a grammar assuming the alphabet is Σ = {a b}.

34.1 Grammar Name and Alphabet

A descriptive name for the grammar is MULT3-as. As specified by the problem statement, the alphabet for L(MULT3-as) is Σ = {a b}.

34.2 Syntactic Categories

The starting syntactic category generates all words in L(MULT3-as). We define it as follows:

S = words where the number of a is 3n, starting nonterminal

S may start by generating an a or a b. If it generates a b, then it must still generate a word in which the number of a is a multiple of 3. That is, generating a b means that an S must also be generated. If S generates an a, then it must also generate a word in which the number of as is a multiple of 3 plus 2. This is a new type of word, and its syntactic category, B, is described as follows:

B = words where the number of a is 3n + 2

If B generates a b, it must still generate a word with 3n+2 as. Therefore, generating a b means that a B must also be generated. If B generates an a, then a word with 3n+1 as must be generated. This is a new type of word, and its syntactic category is defined as follows:

C = words where the number of a is 3n + 1

If C generates a b, it must still generate a word with 3n+1 as. Therefore, generating a b means that a C must also be generated. If C generates an a, then a word with 3n as must be generated. That is, generating an a means that an S must also be generated.

There are no more syntactic categories to define, and we may move to the next step of the design recipe.

34.3 The Production Rules

The production rules are developed one nonterminal at a time. Start, for example, with `S`. According to the syntactic category, `S` must generate a word with `3n` as. The empty word is in `L(MULT3-as)`. This means `S` must generate `EMP`. `S` can also start by generating an `a` or a `b`. If it generates an `a`, then a word with `3n+2` as must also be generated to end with a word that has `3n` as. If it generates a `b`, then a word with `3n` as must still be generated to end with a word that has `3n` as. Based on the meaning of each syntactic category, the needed production rules are:

```
(list 'S ARROW EMP)
(list 'S ARROW 'aB)
(list 'S ARROW 'bS)
```

`B` can start by generating an `a` or a `b`. If it generates an `a`, then a word with `3n+1` as must also be generated to end with a word that has `3n+2` as. If it generates a `b`, then a word with `3n+2` as must still be generated. Based on the meaning of each syntactic category, the needed production rules are:

```
(list 'B ARROW 'aC)
(list 'B ARROW 'bB)
```

`C` can start by generating an `a` or a `b`. If it generates an `a`, then a word with `3n` as must also be generated. If it generates a `b`, then a word with `3n+1` as must still be generated. Based on the meaning of each syntactic category, the needed production rules are:

```
(list 'B ARROW 'aC)
(list 'B ARROW 'bB)
```

34.4 Unit Tests

The tests are written using `MULT3-as` to derive words that are and that are not in `L(MULT3-as)`. Sample test for `MULT3-as` are:

```
;; Tests for MULT3-as
(check-equal? (grammar-derive MULT3-as '(b b a b b))
              "(b b a b b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '(b b a b b a))
              "(b b a b b a) is not in L(G).")
(check-equal?
  (grammar-derive MULT3-as '(b b a b a b a a b))
  "(b b a b a b a a b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '())
              '(S -> ϵ))
```

```
(check-equal?
  (grammar-derive MULT3-as '(a a a))
  '(S -> aB -> aaC -> aaaS -> aaa))
(check-equal?
  (grammar-derive MULT3-as '(b b a a b a b b))
  '(S -> bS -> bbS -> bbaB -> bbaaC -> bbaabC -> bbaabaS
    -> bbaababS -> bbaababbS ->  bbaababb))
```

34.5 Grammar Implementation

The grammar implementation uses `make-rg` and the results of steps 1–3 of the design recipe for grammars:

```
(define MULT3-as (make-rg '(S B C)
                          '(a b)
                          `((S ,ARROW ,EMP)
                            (S ,ARROW aB)
                            (S ,ARROW bS)
                            (B ,ARROW aC)
                            (B ,ARROW bB)
                            (C ,ARROW aS)
                            (C ,ARROW bC))
                          'S))
```

34.6 Run the Tests

Run the tests and make sure they all pass. In addition, use `grammar-test` to visually inspect the result of trying to derive randomly generated words. The following displays the results obtained using ten randomly generated words:

```
> (grammar-test MULT3-as 10)
'(((b a a a) (S -> bS -> baB -> baaC -> baaaS -> baaa))
  ((b a a a a) "(b a a a a) is not in L(G).")
  ((a a) "(a a) is not in L(G).")
  ((a b a b) "(a b a b) is not in L(G).")
  ((a b) "(a b) is not in L(G).")
  (() (S -> ϵ))
  ((b) (S -> bS -> b))
  ((b b) (S -> bS -> bbS -> bb))
 ((b a b a) "(b a b a) is not in L(G).")
```

```
((b a a b a b)
 (S
  ->
  bS
  ->
  baB
  ->
  baaC
  ->
  baabC
  ->
  baabaS
  ->
  baababS
  ->
  baabab)))
```

You can observe that the correct result is obtained for each test word. This further validates that the grammar is correctly implemented.

1 Let Σ={`a b c`}. Design and implement a regular grammar for all words that have an even number of `b`. Make sure to follow all the steps of the design recipe for grammars.

2 Let Σ = {`a b`}. Design and implement a regular grammar for all words that have an even number of `b`s and an odd number of `a`s. Make sure to follow all the steps of the design recipe for grammars.

3 Let Σ = {`a b`}. Design and implement a regular grammar for all words in which every `b` is immediately preceded by an `a`. Make sure to follow all the steps of the design recipe for grammars.

4 Let Σ = {`a b`}. Design and implement a regular grammar for all words that contain `(a b b a)`. Make sure to follow all the steps of the design recipe for grammars.

5 Let Σ = {`a b`}. Design and implement a regular grammar for all words that do not contain `(a b a)`. Make sure to follow all the steps of the design recipe for grammars.

35 Regular Grammars and Regular Languages

Despite being called *regular* grammars, do these grammars generate a regular language? That is, is every language generated by a regular grammar regular, and can every regular language be generated by a regular grammar? If so, given a regular grammar, we ought to be able to construct a finite-state machine or regular expression from it and vice versa.

It is, indeed, the case, and we may state the following theorem:

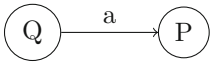
Theorem 1 *L is regular* ⇔ *L is generated by a regular grammar.*

Proof The theorem follows from Theorems 2 and 3 proven below. □

35.1 Constructing a Regular Grammar from a `dfa`

Theorems 7 and 8 establish the equivalence of regular expressions and `ndfa`s. Theorem 4 establishes the equivalence of `ndfa`s and `dfa`s. Therefore, if `L` is regular, we know that there must be a `dfa` that decides `L`. If the language of a regular grammar is regular, then there must be a `dfa` that decides it.

Consider the problem of building a regular grammar, `R`, from, `D`, a `dfa` such that `L(R) = L(D)`. `D`'s transition rules always consume an element of the alphabet. We can view a transition rule as follows:

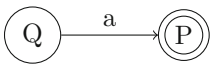


If `D` consumes `a` to move from `Q` to `P`, then `R` must have a production rule to produce such an `a`. To write such a production rule, however, it is necessary to know what may be substituted (i.e., the left-hand side of the production rule) and what to substitute it with (i.e., the right-hand side of the production rule). Clearly, the `a` must be part of the right-hand side of the production rule. We do not know what `D` may consume after reaching `P`, but whatever it is, it must be generated by production rules obtained from transition rules starting at `P`. This means that for the transition rule above, the `a` and anything read after reaching `P` must be generated. Recall that `Q` and `P` represent an invariant property of what the machine has consumed. In a grammar, we may think of `Q` and `P` as the type of word that must still be produced. That is, the states of `D` are the nonterminals of the regular grammar. From `Q`, an `a` and whatever is produced by `P` must be produced. Therefore, for `D`'s rule above, the needed production rule is:

```
(list 'Q ARROW 'aP)
```

This discussion assumes that `D`'s states are represented using a single capital letter in the Roman alphabet. Be aware that this limits the number of states `D` may have to 26 and that `FSM`'s `DEAD` state may not be a state in `D`.

The production rules of a regular grammar cannot all have the form above. If all production rules had that form, then the language of the grammar is empty. No words can be produced because every production rule produces a nonterminal. We need production rules that do not produce a nonterminal. Consider the following transition rule in `D`:



For such a transition rule, going into a final state, an `a` must be produced. After the `a`, however, nothing may need to be produced because `P` is a final state and the machine may accept if there is nothing left in the input. In such a case, only the `a` needs to be produced by the grammar. There may be, of course, more input for `D` to consume, and in this case, a production rule that generates `P`, as above, is also needed. Therefore, for transition rules into a final state, two production rules are needed as follows:

```
(list 'Q ARROW 'a)
(list 'Q ARROW 'aP)
```

Finally, we may observe that nothing may have to be produced. This can only occur if `D`'s starting state, `S`, is a final state. If such is the case, the following production rule is needed:

```
(list 'S ARROW EMP)
```

35.1.1 Implementation

The previous observations suggest an algorithm for transforming a list of `dfa` rules into production rules. For each `dfa` rule, `(Q a P)`, there are either one or two production rules generated. The production rule `(Q ARROW aP)` is always generated. To transform `(a P)` into a symbol, `FSM`'s `los->symbol` is used. If `P` is a final state, then `(Q ARROW a)` is also generated. This transformation may be achieved by `append-map`ing over a list of `dfa` rules. The mapped function takes as input a `dfa` rule and returns a list of production rules of either length 1 or 2. The function to compute the production rules may be implemented as follows:

```
;; (listof dfa-rule) (listof state) → (listof rg-rule)
;; Purpose: Generate production rules for the given
;;          dfa-rules and the given final states
(define (mk-prod-rules mrules mfinals)
 (append-map
   (λ (r)
     (if (not (member (third r) mfinals))
         (list (list (first r) ARROW (los->symbol (rest r))))
         (list (list (first r) ARROW (second r))
               (list (first r) ARROW (los->symbol (rest r))))))
   mrules))
```

```
;; Tests for mk-prod-rules
(check-equal? (mk-prod-rules '() '(F G)) '())
(check-equal? (mk-prod-rules '((S a F) (S b R)
                               (R a G) (R b R)
                               (G a G) (G b G))
                             '(F G))
              '((S -> a) (S -> aF) (S -> bR)
                (R -> a) (R -> aG) (R -> bR)
                (G -> a) (G -> aG) (G -> b)  (G -> bG)))
```

Observe that the test clearly illustrate the production rules generated for each `dfa` rule.

The function to convert a `dfa`, `M`, into a regular grammar, `rg`, takes as input a `dfa` and returns an `rg`. It assumes all machine states are represented by a single capital letter in the Roman alphabet. It locally defines the grammar's nonterminals as the given machine's state, the grammar's alphabet as the machine's alphabet, the grammar's starting nonterminal as the machine's start state, and the grammar's production rules as the result of calling `mk-prod-rules` with the machine's rules and final states. If the machine's starting state is a final state, then a production rule to generate `EMP` from the starting state is added to the production rules. The function based on this design is displayed in Fig. 51. The tests are written using the `dfa` for the language containing the words with an even number of `a`s and an odd number of `b`s from Sect. 24.5 and the following `dfa` for Σ^*:

```
;; L = (a U b U c)*
(define SIGMA* (make-dfa '(S)
                         '(a b c)
                         'S
                         '(S)
                         '((S a S)
                           (S b S)
                           (S c S))
                         'no-dead))
```

Each of these machines is converted into a regular grammar. For a `w`$\in$`L(M)`, the last element in a derivation returned by `grammar-derive` must be a symbol representing `w`, and `M` must accept `w`. For `w`$\notin$`L(M)`, `grammar-derive` must return a string, and `M` must reject `w`.

Run the tests and make sure they all pass. With the cautious confidence provided by having all tests pass, we proceed to argue the correctness of the constructor.

Fig. 51 The function to convert a `dfa` into a regular grammar

```
;; dfa → rg
;; Purpose: Build a rg for the language of the given dfa
;; Assume: States in the given dfa are represented by a single capital letter
(define (dfa2rg m)
  (let* [(nts (sm-states m))
         (sigma (sm-sigma m))
         (startnt (sm-start m))
         (prules (if (member (sm-start m) (sm-finals m))
                     (cons (list (sm-start m) ARROW EMP)
                           (mk-prod-rules (sm-rules m) (sm-finals m)))
                     (mk-prod-rules (sm-rules m) (sm-finals m))))]
    (make-rg nts sigma prules startnt)))

;; Tests for dfa2rg
(define SIGMA*-rg (dfa2rg SIGMA*))
(define EA-OB-rg  (dfa2rg EVEN-A-ODD-B))

(check-equal? (eq? (last (grammar-derive SIGMA*-rg '())) EMP)
              (eq? (sm-apply SIGMA* '()) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(a b c)))
                   (los->symbol '(a b c)))
              (eq? (sm-apply SIGMA* '(a b c)) 'accept))
(check-equal? (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))
                   (los->symbol '(c c a b a c)))
              (eq? (sm-apply SIGMA* '(c c a b a c)) 'accept))

(check-equal? (string? (grammar-derive EA-OB-rg '(a b)))
              (eq? (sm-apply EVEN-A-ODD-B '(a b)) 'reject))
(check-equal? (string? (grammar-derive EA-OB-rg '(a a b a)))
              (eq? (sm-apply EVEN-A-ODD-B '(a a b a)) 'reject))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b)))
                   (los->symbol '(b)))
              (eq? (sm-apply EVEN-A-ODD-B '(b)) 'accept))
(check-equal? (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))
                   (los->symbol '(b a a b b)))
              (eq? (sm-apply EVEN-A-ODD-B '(b a a b b))
```

35.1.2 Correctness Proof

Theorem 2 *L is regular ⇒ L is generated by a regular grammar.*

Proof

Assume L is regular.

Let M = (make-dfa S Σ Z F δ) that decides L, and let G be the regular grammar returned by (dfa2rg M). We must show that w∈L(M) ⇔ w∈L(G) and that w∉L(M) ⇔ w∉L(G).

$w \in L(M) \Leftrightarrow w \in L(G)$

($\Rightarrow$) Assume $w \in L(M)$.
This means that M performs the following computation: (w Z) $\vdash^*_M$ (() K), where $K \in F$. By construction of G, for every rule, (C a D), in M, there is a corresponding production rule ($C \rightarrow aD$). In addition, if D is a final state, then ($C \rightarrow a$) is also a production rule. This means that every element consumed by M's computation is generated by a derivation using G simulating M's execution. If M moves to a final state and accepts, then the derivation only generates a terminal symbol that is the same symbol consumed by M. In this manner, it generates the same word M accepts. Finally, if nothing is consumed and M accepts, then the derivation using G generates EMP. This means there is a derivation using G that generates a word accepted by M. Thus, $w \in L(G)$.

($\Leftarrow$) Assume $w \in L(G)$.
This means that G can derive w. By construction of G, every derivation step of G simulates a step taken by M in a computation that consumes w and reaches a final state. Therefore, $w \in L(M)$.

$w \notin L(M) \Leftrightarrow w \notin L(G)$

($\Rightarrow$) Assume $w \notin L(M)$.

This means that M's computation on w is (w Z) $\vdash^*_M$ (() K), where $K \notin F$. By construction of G, for every rule, (C a D), in M, there is a corresponding production rule ($C \rightarrow aD$). That is, a derivation using G simulates the computation done by M. Let (L a K) be the last transition rule M uses during its computation on w. This means that the last production rule G uses is ($L \rightarrow aK$). Observe that the symbol produced still represents the existence of a nonterminal and, thus, cannot equal w. Given that G is simulating a `dfa` and K is not a final state of M, there is no other possible set of derivation steps on w. Thus, $w \notin L(G)$.

($\Leftarrow$) Assume $w \notin L(G)$.

This means that there is no derivation of w using G. By construction, a derivation using G is simulating the computation of M on w. Given that M is deterministic, there is only one computation possible. Thus, $w \notin L(M)$. □

6 The proof of Theorem 2 states twice that *a derivation using `G` simulates the computation done by `M`.* This suggests that the proof can be even more rigorous by demonstrating that every transition made by `M` on `w` is simulated in a derivation of `w` using `G`. Prove Theorem 2 by induction on the number of transitions in `M`'s computation on `w`.

35.2 Constructing an `ndfa` from a Regular Grammar

We now need to convert `G=(make-rg N` Σ `P S)` into, `M`, a finite-state machine that decides `L(G)`. The constructor for a regular grammar from a `dfa` to a `rg` provides some insights into this problem. That constructor informs us that a regular grammar can simulate a computation carried out by a `dfa`. It is, therefore, plausible that a finite-state machine can simulate a derivation.

Inspired in `dfa2rg`'s design, `G`'s nonterminals may be represented as states in `M` that have the same name. Σ is `M`'s alphabet and `S` is `M`'s starting state.

If `M` is to simulate a derivation under `G`, then `M` must accept when a simple production rule is used. A simple production rule is defined as follows:

`I` $\rightarrow$ `i, where i`$\in\{\{\epsilon\}\cup\Sigma\} \wedge$ `I`$\in$`N`

`M` must also have a transition rule for every compound production rule in `G`. A compound production rule is defined as follows:

`I` $\rightarrow$ `iJ, where i`$\in\Sigma \wedge$ `I,J`$\in$`N`

To simulate a simple production rule, `M` shall have, `Z`, a unique state that is also the single final state. The transformation of a simple production rule is as follows:

`I` $\rightarrow$ `i  -->  (I i Z)`

Observe that this means that `M` accepts when a derivation produces a word and that the `M`'s set of states are `G`'s nonterminals and `Z`.

To simulate a compound production rule, it is converted to a transition rule as follows:

`I` $\rightarrow$ `iJ  -->  (I i J)`

If a derivation step produces from `I` a terminal `i` and a nonterminal, `J`, then `M` moves from state `I` to state `J` on an `i`. Note that defining the transition rules in this manner means that an `ndfa` is constructed (e.g., there are no transitions out of `Z`).

To illustrate how the constructor ought to work, consider converting the following regular grammar into an `ndfa`:

```
;; L = a* U b*
(define a*Ub*-rg (make-rg '(S A B)
                          '(a b)
                          `((S ,ARROW ,EMP)
                            (S ,ARROW aA)
                            (S ,ARROW bB)
                            (S ,ARROW a)
                            (S ,ARROW b)
                            (A ,ARROW aA)
                            (A ,ARROW a)
                            (B ,ARROW bB)
                            (B ,ARROW b))
                          'S))
```

To build `M=(make-ndfa S` Σ `A F` δ`)`, a fresh state, `Z`, is generated for its final state. This means that we may now define four of `M`'s components:

```
S = (cons 'Z '(S A B))
Σ = '(a b)
A = 'S
F = (list 'Z)
```

To generate the transition relation, the production rules may be partitioned into simple and compound production rules:

```
I → i rules              I → iJ rules
  (S → ϵ)                  (S → aA)
  (S → a)                  (S → bB)
  (S → b)                  (A → aA)
  (A → a)                  (B → bB)
  (B → b)
```

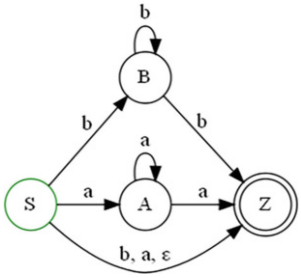
The simple production rules produce the transition rules into the final state:

```
(S ϵ Z)  (S a Z)  (S b Z)  (A a Z)  (B b Z)
```

The compound production rules produce the rest of the transition rules:

```
(S a A)  (S b B)  (A a A)  (B b B)
```

The transition diagram for the resulting `ndfa` is:



It is not difficult to see that it decides `L = a`* $\cup$ `b`*.

35.2.1 Implementation

Based on the outlined design idea, we shall implement a constructor that converts a `rg` to an `ndfa`. To test the constructor, the following sample grammars are used:

```
;; Sample rg
;; L = (a U b U c)*
(define SIGMA*-rg (dfa2rg SIGMA*))

;; L = w | w has an even number of a and an odd number of b
(define EA-OB-rg (dfa2rg EVEN-A-ODD-B))

;; L = a* U b*
(define a*Ub*-rg
  (make-rg
    '(S A B)
    '(a b)
    `((S ,ARROW ,EMP) (S ,ARROW aA) (S ,ARROW bB) (S ,ARROW a)
      (S ,ARROW b)    (A ,ARROW aA) (A ,ARROW a)  (B ,ARROW bB)
      (B ,ARROW b))
    'S))

;; L = a aba
(define a-aba-rg
  (make-rg
    '(S A B)
    '(a b)
    `((S ,ARROW a)  (S ,ARROW aA)  (A ,ARROW bB) (B ,ARROW a))
    'S))
```

The first two are used to illustrate that converting from a given `dfa` to a `rg` and then to an `ndfa` constructs a machine that decides the same language as the given `dfa`. The third is the `rg` used to illustrate the algorithm above. Finally, the fourth is for a finite language that clearly illustrates that simulating simple production rules lead the machine to its final state.

To test the constructor, `rg2ndfa`, `ndfa`s are built from the four sample `rg`s. The tests are displayed in Fig. 52. For each `ndfa`, the tests illustrate that when the `ndfa` rejects a given word, then the given word cannot be derived using the corresponding grammar and that when the `ndfa` accepts a given word, then the given word is derived using the corresponding grammar. For `SIGMA*2`, there are no tests for rejected words because it accepts all possible words. Given that `SIGMA*2` and `EA-OB` are constructed from a `rg` that is constructed from a `dfa`, the equivalence of `SIGMA*2` and `EA-OB` with, respectively, `SIGMA` and `EVEN-A-ODD-B` is tested.

The `rg2ndfa` implementation defines eight local variables to build the components of the `ndfa`:

`final-state`: the generated symbol for the only final state.
`states`: the `ndfa`'s states are the nonterminals of the given grammar and the generated final state.
`sigma`: the `ndfa`'s alphabet is the given grammar's alphabet.
`start`: the `ndfa`'s start state is the given grammar's starting nonterminal.
`finals`: the `ndfa`'s final states only contain the generated final state.
`simple-prs`: the given grammar's simple production rules.
`cmpnd-prs`: the given grammar's compound production rules.
`rules`: the `ndfa`'s transition relation.

The simple production rules are obtained by filtering the given grammar's production rules. The function used to filter the production rules converts the right-hand side of the given production rule into a list and checks if its length is 1. The compound production rules are obtained by filtering the given grammar's production rules. The function used to filter the production rules converts the right-hand side of the given production rule into a list and checks if its length is 2. The `ndfa`'s transition rules are obtained by appending transition rules generated from the simple production rules and transition rules generated from the compound production rules. To generate the first set of transition rules, a function is mapped over the simple production rules. This function creates a transition using the left- and right-hand sides of the given production rule and the generated final state. To generate the second set of transition rules, a function is mapped over the compound production rules. This function converts the right-hand side of the given compound production rule into a list and creates a transition using the left-hand side of the given production rule and the two elements in the converted right-hand side. The result of this implementation strategy is displayed in Fig. 53.

Fig. 52 Tests for `rg2ndfa`

```
;; Tests for rg2ndfa
(define SIGMA*2 (rg2ndfa SIGMA*-rg))  (define EA-OB (rg2ndfa EA-OB-rg))
(define a*Ub*   (rg2ndfa a*Ub*-rg))   (define a-aba (rg2ndfa a-aba-rg))

(check-equal? (eq? (sm-apply SIGMA*2 '()) 'accept)
              (eq? (last (grammar-derive SIGMA*-rg '())) EMP))
(check-equal? (eq? (sm-apply SIGMA*2 '(a b c)) 'accept)
              (eq? (last (grammar-derive SIGMA*-rg '(a b c)))
                   (los->symbol '(a b c))))
(check-equal? (eq? (sm-apply SIGMA*2 '(c c a b a c)) 'accept)
              (eq? (last (grammar-derive SIGMA*-rg '(c c a b a c)))
                   (los->symbol '(c c a b a c))))
(check-equal? (sm-testequiv? SIGMA* SIGMA*2) #t)
(check-equal? (eq? (sm-apply EA-OB '(a b)) 'reject)
              (string=? (grammar-derive EA-OB-rg '(a b))
                        "(a b) is not in L(G)."))
(check-equal? (eq? (sm-apply EA-OB '(a a b a)) 'reject)
              (string=? (grammar-derive EA-OB-rg '(a a b a))
                        "(a a b a) is not in L(G)."))
(check-equal? (eq? (sm-apply EA-OB '(b)) 'accept)
              (eq? (last (grammar-derive EA-OB-rg '(b))) (los->symbol '(b))))
(check-equal? (eq? (sm-apply EA-OB '(b a a b b)) 'accept)
              (eq? (last (grammar-derive EA-OB-rg '(b a a b b)))
                   (los->symbol '(b a a b b))))
(check-equal? (sm-testequiv? EVEN-A-ODD-B EA-OB) #t)
(check-equal? (eq? (sm-apply a*Ub* '(a b)) 'reject)
              (string=? (grammar-derive a*Ub*-rg '(a b))
                        "(a b) is not in L(G)."))
(check-equal? (eq? (sm-apply a*Ub* '(a b b a)) 'reject)
              (string=? (grammar-derive a*Ub*-rg '(a b b a))
                        "(a b b a) is not in L(G)."))
(check-equal? (eq? (sm-apply a*Ub* '()) 'accept)
              (eq? (last (grammar-derive a*Ub*-rg '())) EMP))
(check-equal? (eq? (sm-apply a*Ub* '(a a a)) 'accept)
              (eq? (last (grammar-derive a*Ub*-rg '(a a a))) 'aaa))
(check-equal? (eq? (sm-apply a*Ub* '(b)) 'accept)
              (eq? (last (grammar-derive a*Ub*-rg '(b))) 'b))
(check-equal? (eq? (sm-apply a-aba '(b b)) 'reject)
              (string=? (grammar-derive a-aba-rg
                        '(b b)) "(b b) is not in L(G)."))
(check-equal? (eq? (sm-apply a-aba '()) 'reject)
              (string=? (grammar-derive a-aba-rg '()) "() is not in L(G)."))
(check-equal? (eq? (sm-apply a-aba '(a b a)) 'accept)
              (eq? (last (grammar-derive a-aba-rg '(a b a))) 'aba))
(check-equal? (eq? (sm-apply a-aba '(a)) 'accept)
              (eq? (last (grammar-derive a-aba-rg '(a))) 'a))
```

Fig. 53 The `ndfa` constructor from a `rg`

```
;; rg → ndfa
;; Purpose: Build a ndfa for the language of the given regular grammar
(define (rg2ndfa rg)
  (let* [(final-state (generate-symbol 'Z (grammar-nts rg)))
         (states (cons final-state (grammar-nts rg)))
         (sigma (grammar-sigma rg))
         (start (grammar-start rg))
         (finals (list final-state))
         (simple-prs (filter
                      (λ (pr) (= (length (symbol->fsmlos (third pr))) 1))
                      (grammar-rules rg)))
         (cmpnd-prs
          (filter (λ (pr) (= (length (symbol->fsmlos (third pr))) 2))
                  (grammar-rules rg)))
         (rules (append
                 (map (λ (spr)
                        (list (first spr) (third spr) final-state))
                      simple-prs)
                 (map (λ (pr)
                        (let [(rhs (symbol->fsmlos (third pr)))]
                          (list (first pr) (first rhs) (second rhs))))
                      cmpnd-prs)))]
    (make-ndfa states sigma start finals rules)))
```

35.2.2 Correctness Proof

Let `G` be a regular grammar: `G=(make-rg N` Σ `P A)`. Let `M` be the `ndfa` returned by calling `rg2ndfa` with `G`: `M=(rg2ndfa G)=(make-ndfa S` Σ `A '(Z)` δ`)`, where `S = N` $\cup$ {`Z`}. Finally, let `w=`$a_1 \ldots a_n$`K`, where $a_i \in \Sigma$ and `K`$\in$`{N` $\cup$ $\{\epsilon\}$`}`. That is, `w` ends with a nonterminal in `G`, or it is a word in Σ^*.

To prove that the language generated by `G` is regular, we first prove a lemma stating that `G` derives `w` if and only if `M` reaches `Q` by consuming $a_1 \ldots a_n$. `Q` is `M`'s final state if `K`$=\epsilon$. Otherwise, `Q`$\in$`N` (i.e., `Q` is a state representing a nonterminal of `G`).

Lemma 1 *$S \rightarrow^+ w \Leftrightarrow ((a_1 \ldots a_n)\ A) \vdash^+ (()\ Q)$, where $Q=Z$ if w ends with a terminal symbol and $Q \in N$ if w ends with a nonterminal.*

Proof
($\Rightarrow$) Assume S $\rightarrow^+$ w.

We must show that ($a_1 \ldots a_n$ A) $\vdash^+$ (() Q), where Q=Z if w ends with a terminal symbol and Q$\in$N if w ends with a nonterminal. The proof is by induction on, n, the number of steps (i.e., rules used) in the derivation.

Base Case: n=1
Observe that n=1 means that the derivation uses only a single production rule. There are two cases:

If it is a simple production rule, (S $\rightarrow$ a), then w=a. By construction of M, we have that (S a Z)$\in\delta$. Therefore, (a A) $\vdash^+$ (() Z)=(() Q).

If it is a compound production rule, (S $\rightarrow$ aK), then w=aK. By construction of M, we have that (S a K)$\in\delta$. Therefore, (a A) $\vdash^+$ (() K)=(() Q).

Inductive Step:
Assume: S $\rightarrow^+$ w $\Rightarrow$ ($a_1 \ldots a_k$ A) $\vdash^+$ (() Q), where Q=Z if w ends with a terminal symbol and Q$\in$N if w ends with a nonterminal, for n=k.

Show: S $\rightarrow^+$ w $\Rightarrow$ ($a_1 \ldots a_{k+1}$ A) $\vdash^+$ (() Q), where Q=Z if w ends with a terminal symbol and Q$\in$N if w ends with a nonterminal, for n=k+1.

Assume S $\rightarrow^+$ w for n=k +1.

Given that k$\geq$1, k+1>1. This means that the derivation of w is either:

S $\rightarrow \ldots \rightarrow a_1 \ldots a_k U \rightarrow a_1 \ldots a_k a_{k+1}$, where U$\in$N

S $\rightarrow \ldots \rightarrow a_1 \ldots a_k U \rightarrow a_1 \ldots a_k a_{k+1} V$, where U,V$\in$N

By inductive hypothesis, we have:

(($a_1 \ldots a_k a_{k+1}$) A) $\vdash^*$ ((a_{k+1}) U)

If the last production rule used in the derivation is a simple production rule, (U $\rightarrow$ a_{k+1}), then by construction of M, (U a_{k+1} Z)$\in\delta$. Therefore, (($a_1 \ldots a_k a_{k+1}$) A) $\vdash^*$ (() Z)=(() Q).

If the last production rule used in the derivation is a compound production rule, (U $\rightarrow$ a_{k+1}V), then by construction of M, (U a_{k+1} V)$\in\delta$. Therefore, (($a_1 \ldots a_k a_{k+1}$) A) $\vdash^*$ (() V)=(() Q).

($\Leftarrow$) Assume ($a_1 \ldots a_n$ A) $\vdash^+$ (() Q), where Q=Z if w ends with a terminal symbol and Q$\in$N if w ends with a nonterminal.

We must show that S $\rightarrow^+$ w. The proof is by induction on, n, the number of transitions in M's computation.

Base Case: n=1
This means that w ends with a terminal. Thus, M's computation is either:

$$(() \text{ A}) \vdash (() \text{ Z}) \vee ((\text{a}) \text{ A}) \vdash (() \text{ Z})$$

For the first computation, by construction of M, G must have $(\text{S} \rightarrow \epsilon)$. For the second computation, by construction of M, G must have $(\text{S} \rightarrow \text{a})$. Therefore, $\text{S} \rightarrow^{+} \text{w}$.

Inductive Step:
Assume: $((\text{a}_1 \ldots \text{a}_n) \text{ A}) \vdash^{+} (() \text{ Q})$, where Q=Z if w ends with a terminal symbol and Q∈N if w ends with a nonterminal $\Rightarrow \text{S} \rightarrow^{+} \text{w}$, for n=k.

Show: $((\text{a}_1 \ldots \text{a}_n) \text{ A}) \vdash^{+} (() \text{ Q})$, where Q=Z if w ends with a terminal symbol and Q∈N if w ends with a nonterminal $\Rightarrow \text{S} \rightarrow^{+} \text{w}$, for n=k+1.

Assume $((\text{a}_1 \ldots \text{a}_{k+1}) \text{ A}) \vdash^{+} (() \text{ Q})$, where Q=Z if w ends with a terminal symbol and Q∈N if w ends with a nonterminal. Given that $\text{k} \geq 1$, $\text{k}+1 > 1$. This means that M's computation on $(\text{a}_1 \ldots \text{a}_{k+1})$ is:

$$((\text{a}_1 \ldots \text{a}_{k+1}) \text{ A}) \vdash^{*} ((\text{a}_{k+1}) \text{ R}) \vdash (() \text{ Q})$$

By inductive hypothesis, we have:

$$\text{A} \rightarrow^{*} \text{a}_1 \ldots \text{a}_k \text{R}$$

The last transition in M's computation has either Q=Z or Q≠Z. If Q=Z, then, by construction of G, $(\text{R} \rightarrow \text{a}_{k+1}) \in \text{P}$. Therefore, we have:

$$\text{A} \rightarrow^{*} \text{a}_1 \ldots \text{a}_{k+1}.$$

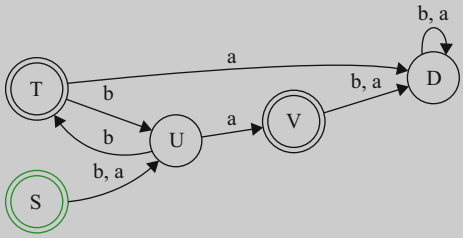
If Q≠Z, then, by construction, of M $(\text{R} \rightarrow \text{a}_{k+1}\text{Q})$. Therefore, we have:

$$\text{A} \rightarrow^{*} \text{a}_1 \ldots \text{a}_{k+1}\text{Q}.$$

Theorem 3 *L is generated by a regular grammar ⇒ L is regular.*

Proof A assume L is generated by a regular grammar. Let G be a regular grammar such that L = L(G), let w∈L, and let M=(rg2ndfa G). By Lemma 1, $\text{S} \rightarrow^{+} \text{w} \Leftrightarrow (\text{w A}) \vdash^{+} (() \text{ Z})$. Given that w is an arbitrary word, M decides L. Thus, L is regular. □

7 Consider the `dfa` displayed in the following transition diagram:



Using `dfa2rg` to build a regular grammar for the language the machine decides results in:

```
(S → ϵ)   (S → aU)   (S → bU)   (U → b)
(U → a)   (U → bT)   (U → aV)   (T → bU)

(U → aV)   (V → aD)   (V → bD)
(T → aD)   (D → aD)   (D → bD)
```

Observe that the production rules in italics can never be part of a derivation for a word in the language of the regular grammar. It is, therefore, unnecessary to generate such production rules. Refine `dfa2rg` so that it builds a grammar without unnecessary production rules. Use `grammar-testequiv` to validate the refined constructor.

8 Prove that your constructor for the previous problem is correct.

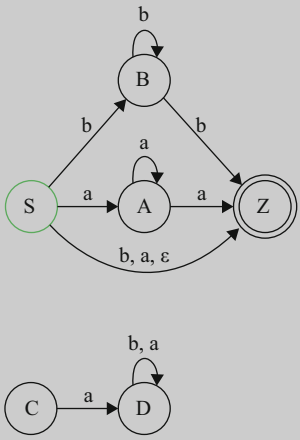
9 The signature for `dfa2rg` is `dfa → rg`. The signature for `rg2ndfa` is `rg → ndfa`. These functions are not inverses of each other. Write a constructor, `rg2dfa`, with the following signature: `rg → dfa`.

10 Prove that your constructor for the previous problem is correct.

11 Consider the following regular grammar production rules:

```
(S → ,EMP)   (S → aA)   (S → bB)   (S → a)   (S → b)
(A → aA)     (B → bB)   (B → b)

(C → aD)     (D → aD)   (D → bD)
```

The transition diagram for an `ndfa` built from a grammar with the above rules is:



Observe that the transition diagram is disconnected. It is unnecessary to generate `C`, `D`, and their transition rules because they can never be part of any computation performed by the machine. Refine `rg2ndfa` to eliminate unnecessary states and their transitions in the `ndfa` returned.

12 Prove that your constructor for the previous problem is correct.

Chapter 9
Languages That Are Not Regular

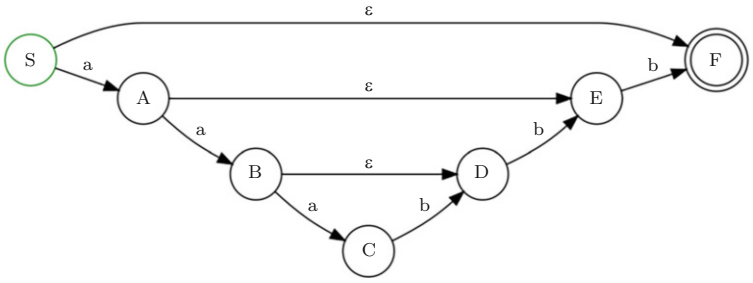
We have a variety of techniques to establish that a language, L, is regular. A regular expression, a dfa, an ndfa, or a rg may be created for L. We have also seen that these artifacts may be used to solve interesting problems. You might already suspect, however, that not all languages in the universe are regular. This belief is likely rooted in the fact that the amount of memory is bounded. For example, the amount of memory in a finite-state machine is bounded by the number of states. It is difficult to see how to prevent loss of knowledge (e.g., what has been read). It would be foolish, however, to dismiss these models as irrelevant to modern computer science. We may not think about it in our day-to-day programming, but the computers we program on also have a finite amount of memory. The finite-state machines that we use on a daily basis are, indeed, quite powerful. Does it surprise you or do you doubt that modern computers are finite-state machines? Just think about it. If a computer has N bits of memory available, then the maximum number of states it can be in is 2^N. If N is in the trillions, then there are a huge number of states our computers can be in. Nonetheless, the number of states is finite just like a dfa or ndfa.

Does a finite amount of memory limit what can be computed? Consider the following language:

$$L = \{a^n b^n \mid n \geq 0\}$$

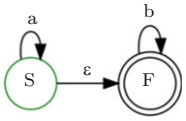
On the surface, it appears to be a rather simple and uninteresting language. What is interesting about a language in which every word has n as followed by n bs? If n is bound to be less than or equal to 3, for example, then implementing an ndfa for the language is fairly straightforward. The transition diagram for the ndfa that decides $L' = \{a^n b^n \mid n \leq 3\}$ is:

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_9



The machine may read 0, 1, 2, or 3 as and then the same number of bs.

The problem with L is that n is a natural number of arbitrary size. How can a finite-state machine read n as and then read n bs? You may argue this is easy by implementing a finite-state machine that has a loop to read n as and then a loop to read n bs:



If we define this machine as a2n-b2n, the following tests pass:

```
;; Tests for a2n-b2n
(check-equal? (sm-apply a2n-b2n '(b b a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '()) 'accept)
(check-equal? (sm-apply a2n-b2n '(a b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a b b)) 'accept)
(check-equal? (sm-apply a2n-b2n '(a a a b b b)) 'accept)
```

Should this give us confidence that the machine decides L? Unfortunately, the answer is an unequivocal no. The tests are not thorough enough. They fail to reveal that the machine is buggy. Consider the following tests:

```
(check-equal? (sm-apply a2n-b2n '(a a)) 'reject)
(check-equal? (sm-apply a2n-b2n '(b)) 'reject)
(check-equal? (sm-apply a2n-b2n '(a a a b)) 'reject)
```

These tests fail. That is, a2n-b2n accepts the words used in the three tests. Clearly, this should not happen.

How can a finite-state machine detect an arbitrary number of as followed by the same number of bs? To do this, the machine needs to remember the number of as read and then make sure it reads the same number of bs. To remember an arbitrary number of as, the machine needs an arbitrary number of states. This, however, is impossible because finite-state machines have a finite number of states and have no other way of remembering how many as were read. This strongly suggests that L is interesting because it is not a regular language. That is, it cannot be decided by a finite-state machine. This

revelation brings us to an interesting question, how can we tell if a language is not regular?

36 The Pumping Theorem for Regular Languages

Cycles in the transition diagram of a finite-state machine and Kleene stars in a regular expression suggest that there is a repetitive structure in a language's words. The cycles or the Kleene star expressions may be repeated 0 or more times. For long-enough words in the language, this repetition must occur one or more times. If a cycle is traversed once, then it moves the machine from a state, `Q`, through 0 or more states to end back at `Q`. If the cycle is traversed once, it may be traversed an arbitrary number of times before moving past `Q` or ending the computation. This means that the symbols read in the loop may be repeated and the resulting word is in the machine's language.

What does this observation suggest? It suggests that for long-enough words in `L(M)`, there must be a repeated subword. We ought to be able to identify the repetition. We may call the repeated subword `y`. A word in the language, therefore, may be written as `w = xyz`, where `x` is the concatenation of the symbols that take the machine to `Q`, and `z` is the concatenation of the symbols that take the machine from `Q` to a final state. Observe that `xyyz`, `xyyyz`, `xyyyyz`, and so on are also in the machine's language. The loop is traversed one or more times. It is also the case that `xz` is in the machine's language. The loop is traversed 0 times. If we generalize, then $\texttt{xy}^{\texttt{i}}\texttt{z}$ is in the machine's language, where $\texttt{i} \geq 0$. That is, if $\texttt{w} \in \texttt{L}$, then we can "pump" up or down on `y` and still have a word that is in `L`. The *Pumping Theorem for Regular Languages* formalizes these observations:

Theorem 1 *For a regular language, L, there is a word length $n \geq 1$ such that any $w \in L$ may be written as $w = xyz$, where $y \neq \epsilon$, $|xy| \leq n$, and $xy^i z \in L$ for $i \geq 0$.*

Before proceeding with the proof, let us be sure we understand what the theorem is stating. It states that a $\texttt{w} \in \texttt{L}$ of length greater than or equal to some positive integer, `n`, may be divided into three parts, `x`, `y`, and `z`, such that `y` is nonempty and may be pumped up or down (i.e., repeated zero or more times) and still remain in the machine's language. Furthermore, it states that the length of `xy` cannot be longer than `n`. That is, `xy` must be at the beginning of `w`. What is this theorem good for? Think about this carefully. For a concrete $\texttt{w} \in \texttt{L}$ that is long enough, we must be able to identify a nonempty `y` that may safely be repeated an arbitrary number of times and still end with a world in `L`. If such a `y` does not exist, then the language is not regular. In other words, the theorem above may be used to prove that a language is not regular.

Proof

Given that L is regular, there is an `dfa`, M=(make-dfa K Σ S F δ), which decides L.

Let n=|K| and w=(a_1 a_2 ... a_n... a_m)∈L, where a_i ∈Σ. The first n steps of M's computation on w are as follows:

$$((a_1\ a_2\ \ldots\ a_n)\ S) \vdash ((a_2 \ldots a_n)\ Q_1) \vdash ((a_3 \ldots a_n)\ Q_2) \vdash \ldots\ (()\ Q_n)$$

Observe that the computation has n+1 configurations (each with the word left to consume and a state). Since M only has n states by the pigeonhole principle, there must be a repeated state in the computation. That is, Q_j=Q_k, where j≠k. This means that $a_j \ldots a_k$ takes the machine from Q_j back to Q_j. That is, part of M's computation is:

$$((a_j \ldots a_k)\ Q_{j-1}) \vdash ((a_{j+1} \ldots a_k)\ Q_j) \vdash^* (()\ Q_j)$$

Given that j<k and M is a `dfa`, ($a_j \ldots a_k$) is not empty. Note that ($a_j \ldots a_k$) may be removed from w or repeated an arbitrary number of times, and the resulting word is still in L. If we define x=($a_1 \ldots a_{j-1}$), y=($a_j \ldots a_k$), and z=($a_{k+1} \ldots a_m$), then ∀i≥0 xy^iz∈L.

Finally, observe that |($a_1 \ldots a_k$)| ≤n because the loop can contain at most all of M's states when x=ϵ. Therefore, |xy| ≤n. □

37 Proving a Language Is Not Regular

To prove that a language is not regular, there are two approaches. The first uses Theorem 1 above, known as the pumping theorem for regular languages. The second uses closure properties from Theorem 1 from Chap. 7.

37.1 Using the Pumping Theorem for Regular Languages

To use the pumping theorem for regular languages, think of it as game against an opponent that will try to demonstrate that the conditions of the theorem can be satisfied. To prove that `L` is not regular, you pick a word, `w`, in the language that is long enough. That is, `w`∈`L` and |`w`| ≥`n`, where `n` is the number of states in the `dfa` that decides `L`. Your opponent tries to find values `x`, `y`, and `z` such that `w = xyz` and xy^iz∈`L`, where `i`≥`0`. If your opponent can find such values, you cannot conclude that `L` is not regular. If your opponent cannot find such values, then you may conclude that `L` is not regular.

To simplify the proof, it is useful to pick w in a manner to minimize the possibilities your opponent has for xy. Recall that |xy| must be less than or equal to n. Try to pick a w that minimizes the choices your opponent has in that window, xy, at the beginning of the word. Fewer choices for y means that fewer arguments must be made about what happens when y is pumped up or down.

37.1.1 L = $\{a^n b^n \mid n \geq 0\}$ Is Not Regular

Theorem 2 *$L = \{a^n b^n | n \geq 0\}$ is not regular*

We shall use Theorem 1.

Proof

Assume L is regular. Let M = (make-dfa K Σ S F δ) be the machine that decides L and let n = |K|. The pumping theorem requires picking a w∈L such that |w| ≥ n. Let w=$a^n b^n$. Clearly, M's computation on w must visit at least one state twice. That is, w is long enough to use the pumping theorem for regular languages. We must now argue that for any valid choice for y, pumping up or down some number of times results in a word not in L. That is, we must show that for some, i≥0 w = $xy^i z \notin L$, such that y $\neq \epsilon$ and |xy| ≤ n.

We can observe that y can only contain as. If it contained any bs, then |xy| would be too long. Thus, y = a^j, where j > 0. We may write w as follows:

$$w = xyz = a^{n-j-r}a^j a^r b^n, \text{ where } x = a^{n-j-r} \wedge z = a^r b^n$$

If we pump up once on y, then we get:

$$w' = a^{n-j-r}a^{2j}a^r b^n = a^{n+j}b^n$$

Clearly, w'∉L. Therefore, the assumption that L is regular is wrong. □

37.1.2 Revisiting L = $\{a^n b^n \mid n \geq 0\}$

You may ask yourself why w=$a^n b^n$ was chosen in the first place. The answer is that it reduced the choices for y to 1. We could have carried out the proof as follows:

Proof

Assume L is regular. Let M = (make-dfa K Σ S F δ) be the machine that decides L and let n = |K|.

Let w=$a^{\frac{n}{2}} b^{\frac{n}{2}}$. Clearly, M's computation on w must visit at least one state twice. We must now argue that for any valid choice for y, there is an i≥0

such that pumping up or down i times results in a word not in L.

The possibilities for y, such that $|xy| \leq n$, are: $y \in a^+$, $y \in a^+b^+$, and $y \in b^+$. For $y \in a^+$, we may write w as follows:

$$w = a^{\frac{n}{2}-r}a^r b^{\frac{n}{2}}, \text{ where } r>0$$

Pumping up once on a^r results in:

$$w' = a^{\frac{n}{2}-r}a^{2r}b^{\frac{n}{2}} = a^{\frac{n}{2}+r}b^{\frac{n}{2}}$$

Clearly, $w' \notin L$.

For $y \in a^+b^+$, we may write w as follows:

$$w = a^{\frac{n}{2}-r}a^r b^s b^{\frac{n}{2}-s}, \text{ where } r,s>0$$

Pumping up once on $a^r b^s$ results in:

$$w' = a^{\frac{n}{2}-r}a^r b^s a^r b^s b^{\frac{n}{2}-s} = a^{\frac{n}{2}}b^s a^r b^{\frac{n}{2}}$$

Clearly, with bs before as, $w' \notin L$.

Finally, for $y \in b^+$, we may write w as follows:

$$w = a^{\frac{n}{2}}b^s b^{\frac{n}{2}-s}, \text{ where } s>0$$

Pumping up once on b^s results in:

$$w' = a^{\frac{n}{2}}b^{2s}b^{\frac{n}{2}} - s = a^{\frac{n}{2}}b^{\frac{n}{2}+s}$$

Clearly, $w' \notin L$.

Given that for all valid choices for y pumping up once results in a word not in L, we may conclude that the assumption that L is regular is wrong. □

As you now see, judicious choice of w simplifies the proof. For L, the proof with a single choice for y is simpler than the proof with three choices for y. Always think carefully about how to choose a word, and structure the argument in your proof.

37.1.3 Sometimes It Is Useful to Pump Down

To prove that a language is not regular, it may be easier to pump down instead of pumping up. We shall use this approach in proving the following theorem.

Theorem 3 *$L = \{a^n b^m \mid n>m\}$ is not regular.*

Proof

Assume L is regular. Let `M = (make-dfa K` Σ `S F` δ`)` be the machine that decides `L` and let `n = |K|`.

Let `w =` $a^{n+1}b^n$. Clearly, `M`'s computation on `w` must visit at least one state twice. We must now argue that there is an `i`$\geq$`0` such that pumping up or down `i` times results in a word not in `L`.

The only possibility for `y`, such that `|xy|` $\leq$`n`, is y = a^p, where p>0. If we pump down once, the resulting word is `w' =` $a^{n+1-p}b^n$. Observe that `n+1-p`$\leq$n. Clearly, `w'` is not in L. Therefore, the assumption that L is regular is wrong.

37.2 Using Closure Properties

The closure properties of regular languages outlined in Theorem 1 from Chap. 7 may also be used to prove that a language is not regular. Let us prove that the language where each word has an equal number of `a`s and `b`s is not regular.

Theorem 4 *$L = \{w \mid w \in (a\ b)^* \wedge w$ has an equal number of `a`s and `b`s$\}$ is not regular*

Proof
Assume L is regular. Consider the following regular language:

```
L' = (concat-regexp
       (kleenestar-reg-exp (singleton-regexp a))
       (kleenestar-reg-exp (singleton-regexp b)))
```

If `L` is regular, then by closure under intersection `L` $\cap$ `L'` is also regular. However, we have that:

`L` $\cap$ `L'` = $a^n b^n$

We know from Theorem 2 that $a^n b^n$ is not regular. Therefore, the assumption that `L` is regular must be wrong. □

To recap, proving that a language is not regular requires a proof by contradiction. Assume that the given language is regular and then show that the assumption leads to a contradiction. The contradiction may be obtained by using the pumping theorem for regular languages or by using closure properties of regular languages.

1 Use the pumping theorem for regular languages to prove that

$$\texttt{L = \{w | w}\in\texttt{(a b)}^* \wedge \texttt{ w has an equal number of as and bs\}}$$

is not regular.

2 Use the pumping theorem for regular languages to prove that the following languages are not regular:

1. $L = \{a^n b^m \mid n \leq m \leq 2n\}$
2. $L = \{ww \mid w \in (a\ b)^*\}$
3. $L = \{b^i a^j \mid i > j\}$
4. $L = \{wcw^R \mid w \in (a\ b)^* \wedge w^R = w \text{ reversed}\}$
5. $L = \{w \mid w \in (a\ b)^* \wedge w \text{ is a palindrome}\}$

3 You just started a new job as a programmer. Your boss asks you to implement a regular expression for the language of balanced parentheses. That is, every word in the language has a matching closing parenthesis for every opening parenthesis. What do you do? Justify your answer.

4 Consider the following language:

$$\texttt{L = \{a}^n\texttt{bc}^n\texttt{ | n}\geq\texttt{0\}} \cup \texttt{\{a}^m\texttt{b}^n\texttt{c}^p\texttt{ | n>0} \wedge \texttt{ m,p}\geq\texttt{0\}}$$

Use closure under intersection and the pumping theorem for regular languages to prove L is not regular.

5 Prove that the English language is not regular. You may find it useful to consider the following sentences:

```
The cat and horse are, respectively, 1 and 2 years old.

The dog, cat, and horse are, respectively, 3, 1, and
2 years old.

The guppy, dog, cat, and horse are, respectively, 7, 3, 1,
and 2 years old.
```

6 Consider the following language:

$$\texttt{L} = \{\texttt{w} = \texttt{xyz} \mid \texttt{w} \in (\texttt{a b})^* \wedge |\texttt{x}| = |\texttt{y}| = |\texttt{z}| \wedge \texttt{z} = \texttt{x with all as substituted by bs and vice versa}\}$$

Prove L is not regular.

7 Consider the following language:

$$\texttt{L} = \{\texttt{w} = \texttt{x}_1\texttt{bx}_2\texttt{bx}_3\texttt{b}\ldots\texttt{bx}_k \mid \texttt{k} \geq 0 \wedge \texttt{x}_i \in \texttt{a}^* \wedge \texttt{x}_i \neq \texttt{x}_j \text{ for } \texttt{i} \neq \texttt{j}\}$$

Prove L is not regular.

8 Prove $\texttt{L} = \{\texttt{a}^n\texttt{b}^m \mid \texttt{n} < \texttt{m}\}$ is not regular.

9 Prove L = {w | w has more as than bs} is not regular.

10 Prove that for M = (make-dfa K Σ S F δ):

$$|\texttt{L(M)}| = \infty \Leftrightarrow \texttt{M accepts a w such that } |\texttt{K}| \leq |\texttt{w}| < 2*|\texttt{K}|$$

11 Prove that $\texttt{L} = \{\texttt{a}^i\texttt{b}^j \mid \texttt{i} \neq \texttt{j}\}$ is not regular using the closure properties of regular languages.

Part III
Context-Free Languages

Chapter 10
Context-Free Grammars

We shall now study how to describe languages that are not regular such as `L` = $\{\mathtt{a}^n\mathtt{b}^n \mid \mathtt{n} \geq 0\}$. Specifically, we shall study *context-free languages*. They are called that because every word in a context-free language is generated by a *context-free grammar*. A context-free grammar, `cfg`, describes the recursive structure of a language. As such, `cfg`s are useful in a variety of applications especially in programming languages and natural language processing.

You are likely familiar with context-free grammars to some degree given that you had to learn how to code in at least one programming language or have taken a course in programming languages or compilers. Usually, the syntax of a programming language is presented as a context-free grammar. For instance, the following context-free grammar specifies how to write definitions in a small arithmetic programming language:

```
<definition> → (define <identifier> <expression>)
             → (define (<identifier> <params>) <expression>)
    <params> → ϵ
             → <identifier> <parameters>
<expression> → <number>
             → <identifier>
             → (+ <expression> <expression>)
             → (- <expression> <expression>)
             → (* <expression> <expression>)
             → (quotient <expression> <expression>)
             → (remainder <expression> <expression>)
```

A context-free grammar ought to remind you of a regular grammar. Just like a regular grammar, it has production rules albeit less restrictive. The syntactic category `<definition>` informs us that there are two subtypes. The first, inside parentheses, has the keyword `define` followed by a variable (i.e., the

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_10

syntactic category `<identifier>`) followed by an expression. The second, inside parentheses, has the keyword `define` followed by a function header containing the name of the function and the names of its parameters, if any, inside parentheses followed by an expression (i.e., the function body). The parameters are zero or more identifiers. There are seven expression subtypes: number, variable reference, and the application of arithmetic operator to two arguments. In essence, a `cfg` tells a programmer what is needed to write a valid program. Put differently, it informs a programmer how to generate a valid program. A context-free grammar, however, does much more than that. It also informs the language implementor of the recursive structure of valid programs and suggests how a program may be interpreted or compiled using structural recursion.

The theory for context-free languages is not as straightforward as the theory for regular languages. It is worth studying them, because they are used to describe the syntax of all modern programming languages and provide the foundation to writing interpreters and compilers. That said, there are important practical consequences. For example, there does not exist a fast (i.e., linear-time) algorithm to decide if a word is in the language of an arbitrary context-free grammar.

38 Context-Free Grammar Definition

The name *context-free* arises from the fact that a production rule for a syntactic category may be applied any time the syntactic category appears in a derivation. It does not matter what is before or after the nonterminal for the syntactic category. Context-free grammars are a type in `FSM`. Now, we can formally define a context-free grammar:

`A context-free grammar is an instance of (make-cfg N` Σ `R S)`

`N` is the set of capital letters in the Roman alphabet representing the nonterminal symbols (i.e., syntactic categories). Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (or terminal symbols). `S` is the starting nonterminal symbol. `R` is the set of production rules. Each production rule is of the form $\texttt{(N} \rightarrow \texttt{(N} \cup \Sigma \cup \epsilon\texttt{)}^{+}\texttt{)}$. That is, there is a single nonterminal on the left hand side of a production rule, and there is a symbol consisting of one or more nonterminals, terminals, or ϵ on the right-hand side of a production rule.

A derivation consists of one or more derivation steps. A derivation step is the application of a production rule and is denoted by $\rightarrow_G$ (or simply $\rightarrow$ if `G` is clear from the context). Zero or more derivation steps are denoted by $\rightarrow_G^*$ (or simply $\rightarrow^*$ if `G` is clear from the context). `L(G)` denotes the language generated by `G`: $\{\texttt{w} \mid \texttt{w} \in \Sigma^* \wedge \texttt{S} \rightarrow_G^* \texttt{w}\}$. Finally, a language, `L`, is context-free if `L = L(G)` for some context-free grammar `G`.

39 L = {a^n b^n | n≥0} Is a Context-Free Language

We prove that L = {a^n b^n | n≥0} is a context-free language by designing and implementing a context-free grammar to generate it. We shall follow the steps of the design recipe for grammars displayed in Fig. 50.

39.1 Steps 1 and 2: Name, Alphabet, and Syntactic Categories

A descriptive name for the grammar is a2nb2n. The alphabet is Σ = {a b}.

The starting syntactic category, S, generates all words that start with n as followed by n bs. How can this be done? Observe that for the first a, there is a matching b at the end of the word. The same is true for the remaining part of the word (i.e., without the first a and last b). This continues to be true until the remaining part of the word is empty. This suggests S ought to generate ϵ. It also suggests that nonempty words in L may be generated from the outside in. That is, generate an a as the first letter in the word, generate a b for the last letter in the word, and use S to generate the rest of the word in the middle. No other nonterminals are needed because no other types of words need to be generated.

39.2 Step 3: The Production Rules

A per our design S ought to generate ϵ. Therefore, the following is a needed production rule:

```
`(S ,ARROW ,EMP)
```

To generate a nonempty word, a production rule to generate the first a, the last b, and a^{n-1}b^{n-1} in the middle is needed. The middle part may be generated using S. Therefore, the needed production rule is:

```
`(S ,ARROW aSb)
```

39.3 Step 4: Tests

Testing context-free grammars in FSM has some practical limitations. The first is that any word used must have a length greater than or equal to 2. The second is that word derivation is computationally intensive. This means that for some context-free grammars, a derivation may take an inordinate

Fig. 54 The `cfg` implementation for `L = {a^n b^n | n≥0}`

```
#lang fsm

;; Syntactic Categories
;;  S = words that start with n a and end with n b

;; L = a^nb^n
(define a2nb2n (make-cfg '(S)
                         '(a b)
                         `((S ,ARROW ,EMP)
                           (S ,ARROW aSb))
                         'S))

;; Tests for a2nb2n
(check-equal? (grammar-derive a2nb2n '(b b b))
              "(b b b) is not in L(G)")
(check-equal? (grammar-derive a2nb2n '(a b a))
              "(a b a) is not in L(G)")
(check-equal? (grammar-derive a2nb2n '(a b))
              '(S -> aSb -> ab))
(check-equal? (grammar-derive a2nb2n '(a a a b b b))
              '(S -> aSb -> aaSbb -> aaaSbbb -> aaabbb))
```

amount of time. As a result, it is most practical to use relative short words in tests but never a word of length 0 or 1.

The tests ought to illustrate the expected result of attempting to derive both words that are and that are not in the grammar's language. The tests are written using `grammar-derive`. How do we know beforehand what a derivation will look like? The answer is that `FSM` always substitutes the leftmost nonterminal first and uses the applicable production rules from top to bottom as they appear in the list of production rules. This helps you write expected values, but does not suffice. A word in a context-free grammar may have several derivations. Therefore, it is possible that the derivation you develop is not the same as the derivation developed by `grammar-derive`. Remember that you can always refine a test if these two derivations are not the same. For `a2nb2n`, we may write tests as follows:

```
;; Tests for a2nb2n
(check-equal? (grammar-derive a2nb2n '(b b b))
              "(b b b) is not in L(G)")
(check-equal? (grammar-derive a2nb2n '(a b a))
              "(a b a) is not in L(G)")
(check-equal? (grammar-derive a2nb2n '(a b))
              '(S -> aSb -> ab))
(check-equal? (grammar-derive a2nb2n '(a a a b b b))
              '(S -> aSb -> aaSbb -> aaaSbbb -> aaabbb))
```

39.4 Steps 5 and 6: Implementation and Testing

Based on the results for the previous steps of the design recipe, the implementation is displayed in Fig. 54.

Run the tests and make sure they all pass. In addition, use `grammar-test` to further test the implementation. The following is a sample running using ten random tests:

```
> (grammar-test a2nb2n 10)
'(((b a b b b a) "(b a b b b a) is not in L(G)")
  (() "The word () is too short to test.")
  ((a a b b) (S -> aSb -> aaSbb -> aabb))
  ((a a) "(a a) is not in L(G)")
  ((a a b b b b a b) "(a a b b b b a b) is not in L(G)")
  ((a a a b a a) "(a a a b a a) is not in L(G)")
  ((a a a b b a a) "(a a a b b a a) is not in L(G)")
  ((a b) (S -> aSb -> ab))
  ((a) "The word (a) is too short to test.")
  ((b b b b) "(b b b b) is not in L(G)"))
```

Be mindful that trying to randomly generate words in the language is not easy. As you can see, most of the time, a word that is too short or that is not in the language is generated. Therefore, it is important for unit tests to be thorough.

40 Practice Designing a `cfg`

Consider the following language:

$$\texttt{L = \{w | w} \in \texttt{(a b)}^* \wedge \texttt{w has more bs than as\}}$$

Is this a context-free language? It is not always easy to tell, but think about how words in the language may be generated. Every word in the language must have at least one more `b` than `a`s. This suggests that at least a `b` must be generated. After that, for every `a` generated, there must be one or more `b`s generated. This is likely something a `cfg` can generate.

40.1 Steps 1 and 2: Name, Alphabet, and Syntactic Categories

A descriptive name for the grammar is `numb>numa`. The alphabet is Σ = {`a` `b`}.

For the needed syntactic categories, we may start by observing that the starting nonterminal must generate all words that have more bs than as. This means it must generate at least a b. After generating a b, words that have more bs than as or equal number of bs and as need to be generated. This is a different type of word, and a different syntactic category is needed for it. We may document the syntactic categories as follows:

```
;; Syntactic Categories
;;  S = words with the number of b > the number of a
;;  A = words with the number of b >= the number of a
```

40.2 Step 3: The Production Rules

According to our design idea, S must at least generate a b. It may also generate words with the number of b greater than or equal to the number of a on either side of the generated b. These observations lead to the following production rules for S:

```
(S ,ARROW b)
(S ,ARROW AbA)
```

Observe that both rules generate words with more bs than as.

To generate words with the number of b greater than or equal the number of a, we can observe that there are three ways this may be done:

- For every a generated, there must be a b generated. The b or the a may be generated first. Before and after each of these, there may be words with the number of b greater than or equal the number of a.
- The empty word may be generated.
- A word starting with a b followed by a word with the number of b greater than or equal the number of a may be generated.

These observations lead to the following production rules:

```
(A ,ARROW AaAbA)
(A ,ARROW AbAaA)
(A ,ARROW ,EMP)
(A ,ARROW bA))
```

Observe that for each rule, the word generated the number of bs is greater than or equal to the number of as.

40.3 Step 4: Tests

The tests ought to illustrate the expected result of attempting to derive both words that are and that are not in the grammar's language. For numb>numa, the following is a sample test suite:

```
;; Tests for numb>numa
(check-equal? (grammar-derive numb>numa '(a b))
              "(a b) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(a b a))
              "(a b a) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(a a a a a))
              "(a a a a a) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(b b b))
              '(S -> AbA -> bA -> bbA -> bbbA -> bbb))
(check-equal?
  (grammar-derive numb>numa '(b b a b a a b))
  '(S -> AbA -> AbAaAbA -> bAaAbA -> bAbAaAaAbA
      -> bAbAaAbAaAaAbA -> bbAaAbAaAaAbA -> bbaAbAaAaAbA
      -> bbabAaAaAbA -> bbabaAaAbA -> bbabaaAbA
      -> bbabaabA -> bbabaab))
(check-equal?
  (grammar-derive numb>numa '(a a a b b b b))
  '(S -> AbA -> AaAbAbA -> aAbAbA -> aAaAbAbAbA
      -> aaAbAbAbA -> aaAaAbAbAbAbA -> aaaAbAbAbAbA
      -> aaabAbAbAbA -> aaabbAbAbA -> aaabbbAbA
      ->  aaabbbbA -> aaabbbb))
```

Observe that for the fourth test, the derivation returned by grammar-derive is:

```
'(S -> AbA -> bA -> bbA -> bbbA -> bbb)
```

A different derivation for bbb is:

```
'(S -> AbA -> bAbA -> bbAbA -> bbbA -> bbb)
```

Therefore, the following test fails:

```
(check-equal? (grammar-derive numb>numa '(b b b))
              '(S -> AbA -> bAbA -> bbAbA -> bbbA -> bbb))
```

This does not mean that the derivation is incorrect. It simply means that grammar-derive finds a different derivation. It does not matter, for our purposes, what derivation is found. What is important is that using the grammar you design and implement, a derivation is found for a word in the language.

Fig. 55 A `cfg` for `L = {w | w∈(a b)* ∧ w has more bs than as}`

```
#lang fsm

;; Syntactic Categories
;;  S = words such that number of b > number of a
;;  A = words such that number of b >= number of a

;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa (make-cfg '(S A)
                        '(a b)
                        `((S ,ARROW b)
                          (S ,ARROW AbA)
                          (A ,ARROW AaAbA)
                          (A ,ARROW AbAaA)
                          (A ,ARROW ,EMP)
                          (A ,ARROW bA))
                          'S))

;; Tests for numb>numa
(check-equal? (grammar-derive numb>numa '(a b))
              "(a b) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(a b a))
              "(a b a) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(a a a a a))
              "(a a a a a) is not in L(G)")
(check-equal? (grammar-derive numb>numa '(b b b))
              '(S -> AbA -> bA -> bbA -> bbbA -> bbb))
(check-equal? (grammar-derive numb>numa '(b b a b a a b))
              '(S -> AbA -> AbAaAbA -> bAaAbA -> bAbAaAaAbA
                  -> bAbAaAbAaAaAbA -> bbAaAbAaAaAbA -> bbaAbAaAaAbA
                  -> bbabAaAaAbA -> bbabaAaAbA -> bbabaaAbA
                  -> bbabaabA -> bbabaab))
(check-equal? (grammar-derive numb>numa '(a a a b b b b))
              '(S -> AbA -> AaAbAbA -> aAbAbA -> aAaAbAbAbA
                  -> aaAbAbAbA -> aaAaAbAbAbAbA -> aaaAbAbAbAbA
                  -> aaabAbAbAbA -> aaabbAbAbA -> aaabbbAbA
                  ->  aaabbbbA -> aaabbbb))
```

40.4 Steps 5 and 6: Implementation and Testing

Based on the results for the previous steps of the design recipe, the implementation is displayed in Fig. 55.

Run the tests and make sure they all pass. In addition, use `grammar-test` to further test the implementation. This is a sample of running five random tests:

```
> (grammar-test numb>numa 5)
'(((a b b b a b b b b)
   (S -> AbA -> AaAbAbA -> AaAbAaAbAbA -> aAbAaAbAbA
    -> abAaAbAbA -> abbAaAbAbA -> abbbAaAbAbA -> abbbaAbAbA
    -> abbbabAbA -> abbbabbA -> abbbabbbA ->  abbbabbbbA
    -> abbbabbbb))
```

```
((a a b b b)
 (S -> AbA -> AaAbAbA -> aAbAbA -> aAaAbAbAbA -> aaAbAbAbA
  -> aabAbAbA -> aabbAbA -> aabbbA -> aabbb))
(() "The word () is too short to test.")
((a b b a b a a b) "(a b b a b a a b) is not in L(G)")
((a b b b b a b)
 (S -> AbA -> AaAbAbA -> aAbAbA -> abAbA -> abAbAaAbA
  -> abbAaAbA -> abbbAaAbA -> abbbbAaAbA -> abbbbaAbA
  -> abbbbabA -> abbbbab)))
```

Be mindful that, for example, the above tests took several minutes to run. If you find testing taking too long, then interrupt execution by clicking the `Stop` button in `DrRacket`.

1 Design and implement a `cfg` for the following language:

$$\texttt{L = \{ww}^R \texttt{ | w}\in\texttt{(a b)}^* \wedge \texttt{w}^R\texttt{=w reversed}\}$$

Make sure to follow all the steps of the design recipe.

2 Design and implement a `cfg` for the following language:

$$\texttt{L = \{w | w}\in\texttt{(a b)}^* \wedge \texttt{w is a palindrome}\}$$

Make sure to follow all the steps of the design recipe.

3 Design and implement a `cfg` for the following language:

$$\texttt{L = \{a}^i\texttt{b}^j \texttt{ | i}\leq\texttt{j}\}$$

Make sure to follow all the steps of the design recipe.

4 Design and implement a `cfg` for the following language:

$$\texttt{L = \{wcw}^R \texttt{ | w}\in\texttt{(a b)}^* \wedge \texttt{w}^R\texttt{=w reversed}\}$$

Make sure to follow all the steps of the design recipe.

5 Design and implement a `cfg` for the following language:

$$\texttt{L = \{a}^i\texttt{b}^j\texttt{c}^k \texttt{ | i,j,k}\geq\texttt{0} \wedge \texttt{(i}\neq\texttt{j or j}\neq\texttt{k)}\}$$

Make sure to follow all the steps of the design recipe.

6 Design and implement a `cfg` for the following language:

$$\texttt{L} = \{\texttt{a}^i\texttt{b}^j\texttt{c}^k\texttt{d}^l \mid \texttt{i,j,k,l} \geq 0 \wedge \texttt{i+j=k+l}\}$$

Make sure to follow all the steps of the design recipe.

7 Design and implement a `cfg` for the following language:

$$\texttt{L} \quad \{\texttt{w} \mid \texttt{w is word with properly balance parenthesis}\}$$

Make sure to follow all the steps of the design recipe. To implement the `cfg`, you will need to represent `(` and `)` with letters such as, respectively, `o` and `c` (for open and close parenthesis).

8 Design and implement a `cfg` for the following language:

$$\texttt{L} = \{\texttt{a}^i\texttt{b}^j \mid \texttt{i} \leq 2\texttt{j}\}$$

Make sure to follow all the steps of the design recipe.

41 All Regular Languages Are Context-Free

We know their existing languages, like $\texttt{L} = \{\texttt{a}^n\texttt{b}^n \mid \texttt{n} \geq 0\}$, that are not regular. An interesting question is: Are all regular languages context-free? If so, it means that the regular languages are a proper subset of the context-free languages. Therefore, any language that can be generated with a regular grammar may also be generated by a context-free grammar. The following theorem establishes that all regular languages are context-free.

Theorem 1 *All regular languages are context-free.*

Proof Assume L is a regular language. This means that there exists a regular grammar:

$$\texttt{G = (make-rg N } \Sigma \texttt{ P S),}$$

such that L = L(G). All the production rules in `P` are of the form:

1. $\texttt{A} \rightarrow \epsilon$
2. $\texttt{A} \rightarrow \texttt{a, where a} \in \Sigma$
3. $\texttt{A} \rightarrow \texttt{aB, where a} \in \Sigma \wedge \texttt{B} \in \texttt{N}$

This means $\texttt{P} \subset (\texttt{N} \rightarrow (\texttt{N} \cup \Sigma \cup \epsilon)^+)$. That is, $\texttt{p} \in \texttt{P}$ is a valid production rule for a context-free grammar. Given that `G`'s `N`, Σ, and `S` may also be used in a `cfg`, we may conclude that `L` is a context-free language. □

Fig. 56 The `cfg` for `L = {w | the number of as in w is a multiple of 3}`

```
(define MULT3-as (make-cfg '(S B C)
                           '(a b)
                           `((S ,ARROW ,EMP)
                             (S ,ARROW aB)
                             (S ,ARROW bS)
                             (B ,ARROW aC)
                             (B ,ARROW bB)
                             (C ,ARROW aS)
                             (C ,ARROW bC))
                           'S))

;; Tests for MULT3-as
(check-equal? (grammar-derive MULT3-as '(b b a b b))
              "(b b a b b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '(b b a b b a))
              "(b b a b b a) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '(b b a b a b a a b))
              "(b b a b a b a a b) is not in L(G).")
(check-equal? (grammar-derive MULT3-as '())
              "The word () is too short to test.")
(check-equal? (grammar-derive MULT3-as '(a a a))
              `(S -> aB -> aaC -> aaaS -> aaa))
(check-equal? (grammar-derive MULT3-as '(b b a a b a b b))
              `(S -> bS -> bbS -> bbaB -> bbaaC -> bbaabC ->
                bbaabaS -> bbaababS -> bbaababbS ->  bbaababb))
```

The theorem above informs us that any regular language may be generated by a context-free grammar. Consider the language of all words in which the number of `a`s is a multiple of 3 from Sect. 34. A `cfg` for this language is displayed in Fig. 56. The only changes from the implementation in Sect. 34 are the constructor used and the expected value of the fourth test (because the tested word is too short).

42 Parse Trees

Consider the language of words containing balanced parenthesis. For each open parenthesis, there must be a matching closing parenthesis. There cannot be a closing parenthesis in the word before its matching opening parenthesis. For example, `()`, `(())`, and `()((()))` are in the language, while `)(`, `(())())`, and `(` are not in the language.

To implement a `cfg` for the language of balanced parenthesis, the opening parenthesis shall be represent by `o`, and the closing parenthesis shall be represented by `c`. To facilitate providing input to, for example, `grammar-derive` and writing tests, a function may be written to convert a string of balanced parenthesis into a list of symbols containing the corresponding `o`s and `c`s. The

Fig. 57 A `cfg` for the language of words with balanced parenthesis

```
#lang fsm

;; string → (listof (o or c)) throws error
;; Purpose: Converts given string of parens to a list of parens
(define (parensstr->los s)
  (map (λ (p)
         (cond [(eq? p #\( ) 'o]
               [(eq? p #\) ) 'c]
               [else (error (format "parensstr->los: non-parens symbol in: ~s"
                                    s))]))
       (string->list s)))

;; Tests for parensstr->los
(check-equal? (parensstr->los "") '())
(check-equal? (parensstr->los "(())()") '(o o c c o c))

;; Syntactic categories
;;   S = words with balanced parenthesis

;; L = {w | w has balanced parenthesis}, where o = ( and c = )
(define BP (make-cfg '(S)
                     '(o c)
                     `((S ,ARROW ,EMP)
                       (S ,ARROW SS)
                       (S ,ARROW oSc))
                     'S))

;; Tests for BP
(check-equal? (grammar-derive BP (parensstr->los ""))
              "The word () is too short to test.")
(check-equal? (grammar-derive BP (parensstr->los "))(("))
              "(c c o o) is not in L(G).")
(check-equal? (grammar-derive BP (parensstr->los "()("))
              "(o c o) is not in L(G).")
(check-equal? (grammar-derive BP (parensstr->los "()"))
              '(S -> oSc -> oc))
(check-equal? (grammar-derive BP (parensstr->los "(())()"))
              '(S -> SS -> oScS -> ooSccS -> ooccS -> ooccoSc -> ooccoc))
```

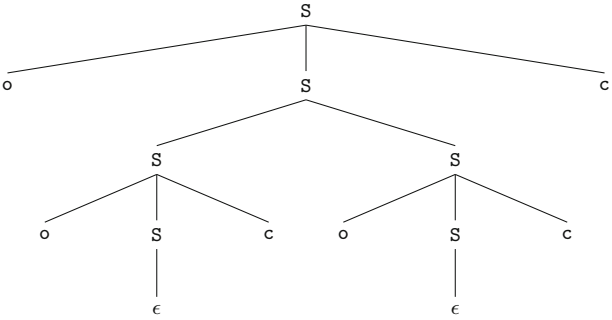
result of following the steps of the design recipe for grammars is displayed in Fig. 57. Note that `#\(` and `#\)`, respectively, represent the characters `(` and `)`.

Consider deriving the `"(()())"`. The result returned by `grammar-derive` is:

```
'(S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc)
```

A derivation may be visualized as a tree as displayed in Fig. 58. Such a visualization is called a parse tree. The leaves are terminal symbols or ϵ. The interior nodes are rooted at a nonterminal. The edges out of an interior node indicate what is generated from the nonterminal. The *yield* of a parse tree is

Fig. 58 The parse tree for `(grammar-derive oococc)`

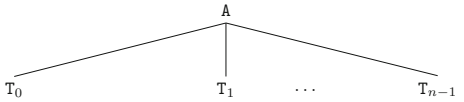


the concatenation of all the leaves from left to right. That is, the yield is the word generated.

A parse tree is formally defined as follows:

- For each $\mathtt{a} \in \Sigma$, the following is a parse tree: $\mathtt{a}$
- For $\mathtt{A} \rightarrow \epsilon$ the following is a parse tree:

- If $\mathtt{T}_0\mathtt{T}_1 \ldots \mathtt{T}_{n-1}$ are the parse trees for $\mathtt{a}_0 \ldots \mathtt{a}_{n-1}$, where $\mathtt{a}_i$ is a nonterminal or a terminal symbol, then for $\mathtt{A} \rightarrow \mathtt{a}_0\mathtt{a}_1 \ldots \mathtt{a}_{n-1}$ the following is a parse tree:



42.1 Similar Derivations

Parse trees represent derivations eliminating irrelevant differences like the order in which production rules are applied. We say that two derivations are similar if they are captured by the same parse tree. For example, the following are similar derivations for `oococc`:

```
S -> oSc -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
S -> oSc -> oSSc -> oSoScc -> oSocc -> ooScocc -> oococc
```

Both of these derivations are captured by the parse tree displayed in Fig. 58. They use the same production rules on the same nonterminals. The only difference is the order in which the production rules are used. The first is a

leftmost derivation. In a leftmost derivation, the leftmost nonterminal is substituted first. The second is a *rightmost derivation*. In a rightmost derivation, the rightmost nonterminal is substituted first. Given a `cfg`, `G = (make-cfg N` Σ `P S)`, every parse tree has a unique leftmost and a unique rightmost derivation. A step in a leftmost derivation is described as follows:

`x` $\overset{L}{\rightarrow}$ `y` $\Leftrightarrow$ `x =` ω`A`β`, y =` $\omega\alpha\beta$`, and A` $\rightarrow$ α, where $\omega \in \Sigma^*$, $\alpha, \beta \in$ `{N` $\cup$ Σ`}`*, and `A`$\in$`N`.

Observe that `A` is the leftmost nonterminal, because ω only contains terminal symbols. A step in the rightmost derivation is described as follows:

`x` $\overset{R}{\rightarrow}$ `y` $\Leftrightarrow$ `x =` β`A`ω`, y =` $\beta\alpha$ ω`, and A` $\rightarrow$ α, where $\omega \in \Sigma^*$, $\alpha, \beta \in$ `{N` $\cup$ Σ`}`*, and `A`$\in$`N`.

Observe that `A` is the rightmost nonterminal, because ω only contains terminal symbols.

Given `G = (make-cfg N` Σ `P S)`, `A`$\in$`N`, and `w`$\in\Sigma^*$, the following statements are equivalent:

- `A` $\rightarrow^*_G$ `w`
- $\exists$ a parse tree with root `A` and yield `w`
- $\exists$ a leftmost derivation `A` $\overset{L}{\rightarrow}{}^*_G$ `w`
- $\exists$ a rightmost derivation `A` $\overset{R}{\rightarrow}{}^*_G$ `w`

That is, a derivation of `w` from `A` using `G` is equivalent to a parse tree rooted at `A` with yield `w`, which is equivalent to a leftmost derivation of `w` from `A` using `G`, which is equivalent to the rightmost derivation of `w` from `A` using `G`. In essence, we have four different forms to describe that from `A` using `G`'s production rules, `w` is obtained.

42.2 Ambiguity

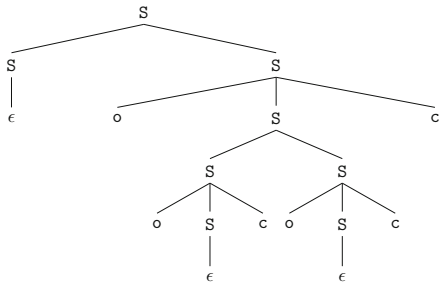
Given a `cfg`, like `BP` in Fig. 57, not all derivations of a word may be similar. For instance, consider the following derivation using `BP`:

```
S -> SS -> S -> oSSc -> ooScSc -> oocSc -> oocoScc -> oococc
```

This derivation is not captured by the parse tree in Fig. 58. It uses, for example, a production rule `S` $\rightarrow$ `SS` that is not present in the parse tree displayed in Fig. 58. The parse tree for this derivation is:

Fig. 59 A `cfg` for simple arithmetic expressions

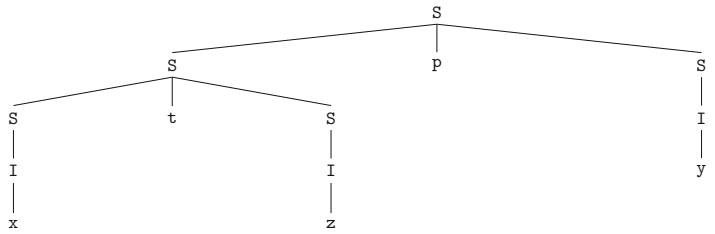
```
#lang fsm

(define AE (make-cfg '(S I)
                     '(p t x y z)
                     `((S ,ARROW SpS)
                       (S ,ARROW StS)
                       (S ,ARROW I)
                       (I ,ARROW x)
                       (I ,ARROW y)
                       (I ,ARROW z))
                     'S))

;; Tests for AE
(check-equal? (grammar-derive AE '(x p x))
              '(S -> SpS -> IpS -> xpS -> xpI -> xpx))
(check-equal? (grammar-derive AE '(x t z))
              '(S -> StS -> ItS -> xtS -> xtI -> xtz))
(check-equal? (grammar-derive AE '(x t z p y))
              '(S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
                  -> xtzpI -> xtzpy))
```



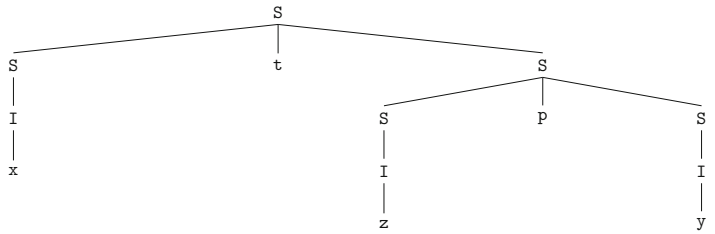
A `cfg`, `G`, is called *ambiguous* if for any `w`∈`L(G)` there are two or more parse trees. Therefore, `BP` is ambiguous. Ambiguity is a problem when meaning must be assigned to a word as done in the implementation of programming languages. For instance, consider the `cfg` for simple addition and multiplication expressions displayed in Fig. 59. In this grammar, `p` stands for `+`, `t` stands for `*`, and `x`, `y`, and `z` are variables. The word in the third test, `'(x t z p y)`, has two derivations that are not similar:

```
S -> SpS -> StSpS -> ItSpS -> xtSpS -> xtIpS -> xtzpS
  -> xtzpI -> xtzpy

S -> StS -> ItS -> xtS -> xtSpS -> xtIpS -> xtzpS -> xtzpI
  -> xtzpy
```

Let us consider the corresponding parse trees. For the first derivation, the parse tree is:



This parse tree suggests that `x` and `z` are multiplied and then their product and `y` are added. This is considered the correct interpretation following the traditional hierarchy of operations. The parse tree for the second derivation is:



This parse tree suggests that first `z` and `y` are added, and then their sum is multiplied by `x`. As you can see, a word may have different meanings when a grammar is ambiguous.

There is a way to disambiguate `AE`. The idea is to first produce the summing subexpressions, if any, and then produce the product subexpressions below them. In this manner, precedence is given to multiplications over additions following the traditional hierarchy of operations. To achieve this, a new syntactic category, `T`, is introduced to generate product subexpressions. A product subexpression may only be generated after summing subexpressions, if any, are generated. The implementation of such a grammar is displayed in Fig. 60.

Fig. 60 An unambiguous `cfg` for simple arithmetic expressions

```
#lang fsm

(define AE2 (make-cfg '(S I T)
                      '(p t x y z)
                      `((S ,ARROW SpS)
                        (S ,ARROW T)
                        (S ,ARROW I)
                        (T ,ARROW TtT)
                        (T ,ARROW I)
                        (I ,ARROW x)
                        (I ,ARROW y)
                        (I ,ARROW z))
                      'S))

;; Tests for AE2
(check-equal? (grammar-derive AE2 '(x p x))
              '(S -> SpS -> IpS -> xpS -> xpI -> xpx))
(check-equal? (grammar-derive AE2 '(x t z))
              '(S -> T -> TtT -> ItT -> xtT -> xtI -> xtz))
(check-equal? (grammar-derive AE2 '(x t z p y))
              '(S -> SpS -> TpS -> TtTpS -> ItTpS -> xtTpS -> xtIpS
                  -> xtzpS -> xtzpI -> xtzpy))
(check-equal? (grammar-derive AE2 '(x p z t z p y))
              '(S -> SpS -> SpSpS -> IpSpS -> xpSpS -> xpTpS -> xpTtTpS
                  -> xpItTpS -> xpztTpS -> xpztIpS -> xpztzpS -> xpztzpI
                  -> xpztzpy))
```

Unfortunately, some context-free languages can only be generated by an ambiguous `cfg`. Such grammars are called *inherently ambiguous*. Fortunately, the grammars developed for programming languages are not inherently ambiguous.

9 Consider the following `cfg`:

```
(make-cfg '(S)
          '(a)
          `((S ,ARROW ,EMP)
            (S ,ARROW aS)
            (S ,ARROW Sa))
          'S)
```

Is this grammar ambiguous? Justify your answer.

10 Using `AE` from Fig. 59, display two different parse trees for:

```
'(x t x p z t z t z p y p y)
```

Pick values for `x`, `y`, and `z` that demonstrate that different values are obtained when plugging into the two different parse trees for this arithmetic expression.

11 Display the rightmost derivation for:

```
'(x t x t x p z t y)
```

12 Consider the following `cfg`:

```
(make-cfg '(S C)
          '(i t e m n f r)
          `((S ,ARROW iCtSeS)
            (S ,ARROW iCtS)
            (S ,ARROW m)
            (S ,ARROW n)
            (C ,ARROW f)
            (C ,ARROW r))
          'S))
```

It represents a grammar for if-then-else statements. Is this grammar ambiguous? Justify your answer.

13 Given a `cfg`, `G`, and a `w∈L(G)` that has two derivations that are not similar. Does this mean there are at least two leftmost derivations for `w`? Justify your answer.

14 Let `G` a `cfg` and let `k>0` be a natural number. Consider the subset of `L(G)` that contains all the words that are derived with `k` or fewer steps using `G`. Prove that this subset is regular.

Chapter 11
Pushdown Automata

We have seen that the members of a context-free language, `L`, may be generated by a context-free grammar. We would also like to have a machine that decides if a given word is a member `L`. Such a machine, of course, cannot be a `ndfa` as we have seen. It is quite natural to ask what features may be added to an `ndfa` to endow it with the power to decide a context-free language.

To help us think about this, let us write a function to decide if a given word, `w`, is a member of $\texttt{L = a}^{\texttt{n}}\texttt{b}^{\texttt{n}}$. One design strategy is to call an auxiliary function that takes as input `w`'s sub-word without the leading `a`s, if any, and `w`'s leading (and at this point unmatched) `a`s as the value of an accumulator. The auxiliary function distinguishes between three conditions:

1. If the given word is empty, then the value from testing if the given accumulator is empty is returned.
2. If the first element of the given word is `a`, then false is returned.
3. Otherwise, return the conjunction of testing if the accumulator is not empty, and check the rest of both the given word and the given accumulator.

The program resulting from this design idea is displayed in Fig. 61.

Observe that the accumulator used in Fig. 61 is accessed in a last-in first-out manner. The program first pushes (i.e., adds) all the `a`s at the beginning of the given word onto the accumulator. The auxiliary function, `check`, pops an `a` for each recursive. In essence, the accumulator is a stack. This suggests that an `ndfa` extended with a stack to remember part of the consumed input may be powerful enough to decide a context-free language.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_11

Fig. 61 A predicate to determine if a word is in L = $a^n b^n$

```
#lang fsm

;; word → Boolean
;; Purpose: Decide if given word is in a^nb^n
(define (is-in-a^nb^n? w)
  ;; word (listof symbol) → Boolean
  ;; Purpose: Determine
  ;; Accumulator Invariant
  ;;   acc = the unmatched as at the beginning of w
  ;; Assume: w in (a b)*
  (define (check wrd acc)
    (cond [(empty? wrd) (empty? acc)]
          [(eq? (first wrd) 'a) #f]
          [else (and (not (empty? acc))
                     (check (rest wrd) (rest acc)))]))
  (check (dropf w (λ (s) (eq? s 'a)))
         (takef w (λ (s) (eq? s 'a)))))

;; Tests for is-in-anbn?
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(b b))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b b))
(check-pred (λ (w) (not (is-in-a^nb^n? w))) '(a b a a b b))
(check-pred is-in-a^nb^n? '())
(check-pred is-in-a^nb^n? '(a a b b))
```

43 Pushdown Automata Definition

An `ndfa` extended with a stack is called a *pushdown automata* (`pda`). The stack is of arbitrary size and provides additional memory beyond the state that is remembered by the control unit. Formally,

> `A (nondeterministic) pushdown automaton, pda, is an instance of:`
>
> `(make-ndpda K` Σ Γ `S F` δ`)`

The inputs to the constructor are defined as follows:

- K: A list of states.
- Σ: An input alphabet.
- Γ: A list of stack symbols.
- S: The starting state.
- F: A list of final (i.e., accepting) states.
- δ: A transition relation.

The transition relation, δ, is a finite subset of:

$$((\mathtt{K} \times (\Sigma \cup \{\mathtt{EMP}\}) \times \Gamma^+ \cup \{EMP\}) \times (\mathtt{K} \times \Gamma^+ \cup \{EMP\})).$$

Each transition rule consists of a triple and a double list. The triple represents the current state, the input to consume, and the stack elements to pop. If no elements are popped, `EMP` is used. In a pop list of length `n`, the leftmost element must match the topmost element on the stack, and the rightmost element must match the $\texttt{n}^{\texttt{th}}$ stack element from the top. If popped sequentially, then the leftmost element is popped first, and the rightmost element is popped last. The double represents the destination state and the stack elements to push. If no elements are pushed, `EMP` is used. In a push list of length `n`, the leftmost element becomes the topmost of the pushed elements, and the rightmost element becomes the bottommost of the pushed elements. If pushed sequentially, then the first element pushed is the rightmost element, and the last element pushed is the leftmost. The machine first pops and then pushes before changing states. In summary, the use of `((A a p) (B g))` means that the machine is in state `A`, reads `a` from the input tape, pops `p` off the stack, pushes `g` onto the stack, and moves to `B`. The machine is nondeterministic, and therefore, `a`, `p`, and `g` may be `EMP`. For instance, `((P EMP EMP) (Q j))` is a push operation that does not consult the input tape nor the stack, and `((P EMP j) (Q EMP))` is a pop operation that does not consult the input tape nor the stack.

A `pda` configuration is a member of $(\texttt{K} \times \Sigma^* \times \Gamma^*)$. It represents the machine's current state, the remaining unread input, and the contents of the stack. For instance, `(S (b b) (a b c))` means that the `pda` is in state `S`, `b b` is the remaining unread input, and the stack contains `a b c`, where `c` is the topmost stack element and `a` is the third element on the stack. A computation step moves the machine from a starting configuration to a new configuration using a single rule denoted as:

$$\texttt{(P, xw, a)} \vdash \texttt{(Q, w, g)}$$

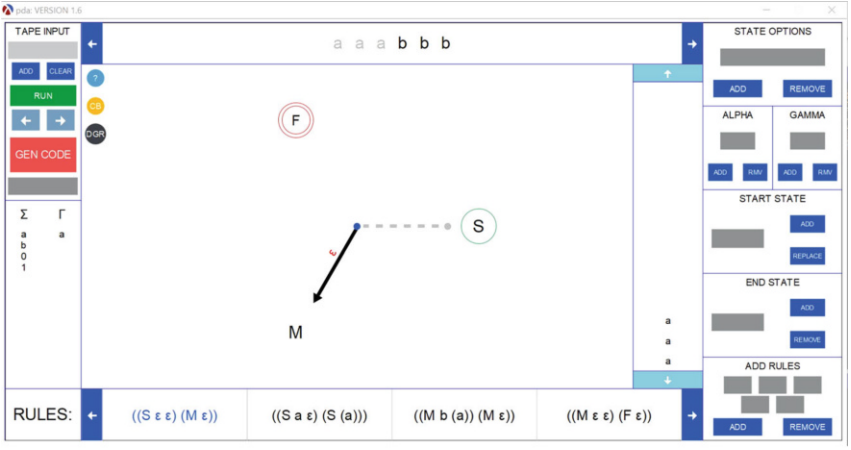
This means that the machine started in state `P`, consumed `x` from the input tape, popped `a` off the stack, pushed `g` onto the stack, and moved to state `Q`. Zero or more steps are denoted using $\vdash^*$. A computation of length `n` on a word, `w`, is denoted by:

$$\texttt{C}_0 \vdash \texttt{C}_1 \vdash \texttt{C}_2 \vdash \ldots \vdash \texttt{C}_n\texttt{,}\ \text{where}\ \texttt{C}_i\ \text{is a}\ \texttt{pda}\ \text{configuration.}$$

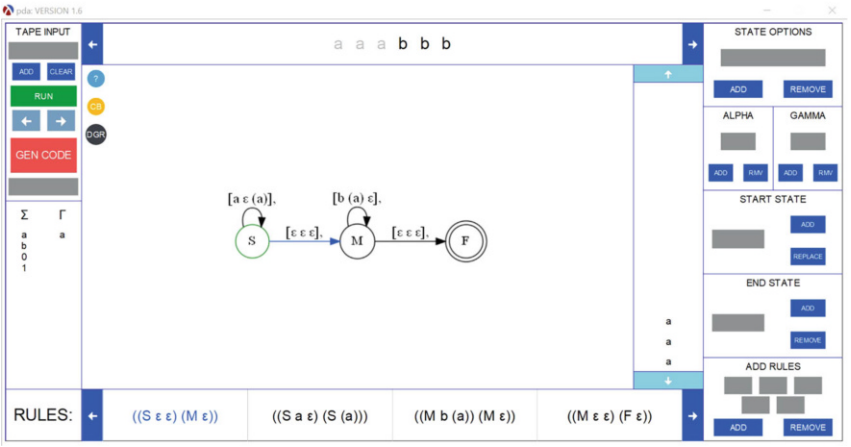
If a `pda`, `M`, starting in the start state consumes all the input and reaches a final state with an empty stack, then `M` accepts. Otherwise, `M` rejects. A word, `w`, is in the language of `M`, `L(M)`, if there is a computation from the start state that consumes and accepts `w`. Like with `ndfa`s, it does not matter that there may be many potential computations that reject `w`. If it exists, a `pda` can sense the right sequence of transitions that lead to accept.

As with `ndfa`s, the computation of words that are accepted may be visualized using `sm-showtransitions` or `sm-visualize`. Figure 62 displays the control view and the transition diagram view for a `pda`. In both views, the stack is to the right of the machine view before the column to perform machine edits. The stack alphabet is displayed in the left column next to the

Fig. 62 Visualization of a `pda`



(a) Control view



(b) Graph view

input alphabet. The configuration displayed in Fig. 62a is `(M (b b b) (a a a))`. In addition, the consumed input is faded out. In Fig. 62b, the same configuration is displayed, and the labels on the edges contain the read element, the popped elements, and the pushed elements.[10]

A state invariant predicate may be associated with each state in the same manner as done for finite-state automatons. For `pda`s, an invariant predicate has two inputs: the consumed part of the word on the input tape and the stack. It must test and relate the invariant conditions for and between the consumed input and the stack.

[10] Currently in `FSM`, `EMP` $= \epsilon$.

44 A pda for L = a^n b^n

As a first example illustrating the design process for pdas, let us design a machine to decide L = a^n b^n. We shall follow the steps of the design recipe for state machines displayed in Fig. 20 in Sect. 23. We must first think about how a stack may be used to decide if a word is in L. We know that for every a in the first half of the word, there must be a matching b in the second half of the word. A stack may be used to accumulate the read as. Once the as have been read, the bs may be matched by popping an a for each. After all the input is read, the machine ought to move to a final state. It accepts if the stack is empty (i.e., all the as and bs have been matched). Otherwise, it rejects.

44.1 Name and Alphabets

A descriptive name for the machine is a^nb^n. The input alphabet Σ = '(a b). Given that only as are pushed onto the stack, the stack alphabet, Γ, is '(a).

44.2 Unit Tests

Unite tests are written for both words in L and words not in L. A sample set of tests are:

```
;; Tests for a^nb^n
(check-equal? (sm-apply a^nb^n '(a)) 'reject)
(check-equal? (sm-apply a^nb^n '(b b)) 'reject)
(check-equal? (sm-apply a^nb^n '(a b b)) 'reject)
(check-equal? (sm-apply a^nb^n '(a b a a b b)) 'reject)
(check-equal? (sm-apply a^nb^n '()) 'accept)
(check-equal? (sm-apply a^nb^n '(a a b b)) 'accept)
```

Observe that care is taken to test words not in L that have a prefix for words that are in L.

44.3 Conditions and States

Deciding if a given word is in L can be done in three general steps. In the first, only as have been read from the input tape, and all have been pushed onto the stack. In the second, the read bs have been matched with corresponding

read as on the stack. In the third, the machine moves to a final state and accepts if all as and bs in the input tape have been matched. We may use the starting state, S, to represent that only as have been read and all are pushed onto the stack. This means that the consumed input and the stack must be equal. We may use a different state, M, to represent that the consumed input contains as followed by bs, that the stack only contains as, and that the read bs have been matched with corresponding read as on the stack. How can this third condition be determined? Observe that the read bs have been matched with read as if the number of as in the consumed input equals the number of as on the stack plus the number of read bs. Finally, a single final state, F, is needed to represent that the stack is empty and the consumed input consists of a number of as followed by an equal number of bs. The states may be documented as follows:

```
;; States
;;  S:   ci = a* = stack, start state
;;  M:   ci = (append (listof a) (listof b))
;;     ∧ stack = a*
;;     ∧ |ci as| = |stack| + |ci bs|
;;  F:    ci = (append (listof a) (listof b))
;;     ∧ |stack| = 0
;;     ∧ |ci as|=|ci bs|, final state
```

Recall that for a nondeterministic machine, the invariant properties only need to hold on computations that end with the machine accepting the given word.

44.4 The Transition Relation

To formulate the transition relation, start with the starting state. At the beginning, no as have been read nor pushed onto the stack. Therefore, the consumed input is an empty (listof a), and the consumed input and the stack are equal. That is, the invariant property for S holds. If an a is read, then it needs to be pushed onto the stack, and the machine needs to remain in S. Observe that if this is done, the invariant property of S holds after reading the a. S should not process a b. Instead, when it is time to process a b, the machine can nondeterministically move to M without changing the stack. The needed transition rules are:

```
((S ,EMP ,EMP) (M ,EMP))     ((S a ,EMP) (S (a)))
```

When the machine transitions from S to M, observe that the consumed input consists of an arbitrary number of as followed by 0 bs and that the number of as in the consumed input is equal to the number of a on the stack and the number of bs (i.e., 0) in the consumed input. That is, the invariant properties for M hold. Once in M, the machine only reads bs. When a b is

Fig. 63 The pda for L = $a^n b^n$

```
;; L = {a^nb^n | n >= 0}
;; States
;;  S ci = (listof a) = stack, start state
;;  M ci = (append (listof a) (listof b)) AND
;;    (length ci as) = (length stack) + (length ci bs)
;;  F ci = (append (listof a) (listof b)) and all as and bs matched,
;;    final state
;; The stack is a (listof a)
(define a^nb^n (make-ndpda '(S M F)
                           '(a b)
                           '(a)
                           'S
                           '(F)
                           `(((S ,EMP ,EMP) (M ,EMP))
                             ((S a ,EMP) (S (a)))
                             ((M b (a)) (M ,EMP))
                             ((M ,EMP ,EMP) (F ,EMP)))))

;; Tests for a^nb^n
(check-equal? (sm-apply a^nb^n '(a)) 'reject)
(check-equal? (sm-apply a^nb^n '(b b)) 'reject)
(check-equal? (sm-apply a^nb^n '(a b b)) 'reject)
(check-equal? (sm-apply a^nb^n '(a b a a b b)) 'reject)
(check-equal? (sm-apply a^nb^n '()) 'accept)
(check-equal? (sm-apply a^nb^n '(a a b b)) 'accept)
```

read, it is matched with a read `a` by popping it off the stack. Observe that if this is done, `M`'s invariant properties hold after reading a `b`. The machine may nondeterministically decide to move to `F`. The needed transition rules are:

```
((M b (a)) (M ,EMP))      ((M ,EMP ,EMP) (F ,EMP))
```

If the machine transitions to `F` when the input tape and the stack are empty, then `F`'s invariant properties hold, and the machine may accept. Otherwise, not all the input has been read, or not all stack elements have been matched. In either case, the machine rejects. There is no need to read any input or manipulate the stack. Thus, no transition rules are needed for `F`.

44.5 Machine Implementation and Testing

The implementation of `a^nb^n` is displayed in Fig. 63. Running the program reveals that all the tests pass. To further test the machine, use `sm-test`:

```
> (sm-test a^nb^n 10)
'(((b b b a a a a a) reject)
  ((a b) accept)
  ((b b b a a b a b a) reject)
```

```
((a a a a b) reject)
((a a b) reject)
((b b b b b a a a) reject)
((b) reject)
((a a b b) accept)
((b b b a b b b a) reject)
((b a a a a) reject))
```

44.6 State Invariant Predicates

The next step requires the design and implementation of state invariant predicates. Recall that each receives the consumed input and the stack as arguments. For S, the invariant checks that the consumed input and the stack have the same length and that both only contain as. To check if both only contain as, andmap may be used to simultaneously traverse both. The resulting predicate is:

```
;; word stack → Boolean
;; Purpose: Determine if the given ci and stack are the
;;          same (listof a)
(define (S-INV ci stck)
 (and (= (length ci) (length stck))
      (andmap (λ (i g) (and (eq? i 'a) (eq? g 'a))) ci stck)))

;; Tests for S-INV
(check-equal? (S-INV '() '(a a)) #f)
(check-equal? (S-INV '(a) '()) #f)
(check-equal? (S-INV '(b b b) '(b b b)) #f)
(check-equal? (S-INV '() '()) #t)
(check-equal? (S-INV '(a a a) '(a a a)) #t)
```

For M, the invariant predicate must establish that the consumed input contains as followed by bs, that the stack only contains as, and that the number of as in the consumed input equals the number of as on the stack plus the number of bs in the consumed input. For this, the as at the beginning of the consumed input may be extracted and locally defined. The same is done for all the bs following these as. To establish the needed invariant properties, the consumed input is tested for equality with the as appended with the bs, andmap is used to verify that the stack only contains as, and the length of the as is tested for equality with sum of the length of the bs and the length of the stack. The resulting predicate is:

```
;; word stack → Boolean
;; Purpose: Determine if ci = EMP or a+b+ AND the stack
;;          only contains a AND |ci as| = |stack| + |ci bs|
```

```
(define (M-INV ci stck)
  (let* [(as (takef ci (λ (s) (eq? s 'a))))
         (bs (takef (drop ci (length as))
                    (λ (s) (eq? s 'b))))]
    (and (equal? (append as bs) ci)
         (andmap (λ (s) (eq? s 'a)) stck)
         (= (length as) (+ (length bs) (length stck))))))

;; Tests for M-INV
(check-equal? (M-INV '(a a b) '(a a)) #f)
(check-equal? (M-INV '(a) '()) #f)
(check-equal? (M-INV '(a a a b) '(a a a)) #f)
(check-equal? (M-INV '(a a a b) '(a)) #f)
(check-equal? (M-INV '() '()) #t)
(check-equal? (M-INV '(a) '(a)) #t)
(check-equal? (M-INV '(a b) '()) #t)
(check-equal? (M-INV '(a a a b b) '(a)) #t)
```

For F, the predicate must establish that the consumed input starts with an arbitrary number of as followed by the same number of bs and that all as and bs have been matched. As done for M-INV, the beginning as and the following bs are locally defined. The stack is tested to determine it is empty. The consumed input is tested to establish that it is the extracted as followed by the extracted bs, and the length of the extracted as is tested to establish that it equals the length of the extracted bs. The resulting predicate is:

```
;; word stack → Boolean
;; Purpose: Determine if ci = a^nb^n and stack is empty
(define (F-INV ci stck)
  (let* [(as (takef ci (λ (s) (eq? s 'a))))
         (bs (takef (drop ci (length as))
                    (λ (s) (eq? s 'b))))]
    (and (empty? stck)
         (equal? (append as bs) ci)
         (= (length as) (length bs)))))

;; Tests for F-INV
(check-equal? (F-INV '(a a b) '()) #f)
(check-equal? (F-INV '(a) '()) #f)
(check-equal? (F-INV '(a a a b) '(a a a)) #f)
(check-equal? (F-INV '() '()) #t)
(check-equal? (F-INV '(a b) '()) #t)
(check-equal? (F-INV '(a a b b) '()) #t)
```

Use the invariant predicates in conjunction with the visualization tool to validate that they hold for computations that accept the given word.

44.7 Correctness

The final step asks to establish the correctness of the implemented `pda`. As done for finite-state machines, we start by proving that the state invariant predicates always hold. Subsequently, we prove that `L = L(a^nb^n)`.

We shall use the following definitions:

```
L = a^nb^n        ci = the consumed input  w∈(sm-sigma M)*
F=(sm-finals M) P = a^nb^n
```

44.7.1 Proving State Invariants Hold

Theorem 1 *The state invariants hold when P is applied to w.*

The proof is by induction on, `n`, the number of transitions to consume `w`.

Proof
When P starts, S-INV holds because ci = `'()` and the stack = `'()`. This establishes the base case.
Proof that invariants hold after each transition:

`((S EMP EMP) (M EMP))`: By inductive hypothesis, `S-INV` holds. After using this transition, `M-INV` holds because ci=`'()` and the stack=`'()`.

`((S a EMP) (S (a)))`: By inductive hypothesis, `S-INV` holds. After consuming an `a` and pushing an `a`, P may reach S and by empty transition M. Note that an empty transition into F with a nonempty stack cannot lead to an accept. Therefore, we do not concern ourselves about P making such a transition because P only makes nondeterministic transitions that can lead to accept. That observed, `S-INV` and `M-INV` hold because both the length of the consumed input and of the stack increased by 1, thus, remaining of equal length and because both continue to only contain `a`s.

`((M EMP EMP) (F EMP))`: By inductive hypothesis, `M-INV` holds. After using this transition, `F-INV` holds because ci=`'()` and the stack=`'()`.

`((M b (a)) (M ,EMP))`: By inductive hypothesis, `M-INV` holds. After consuming a `b` and popping an `a`, P may reach M or nondeterministically reach F because it may eventually accept. `M-INV` holds because ci continues to be `a`s followed by `b`s, the stack can only contain `a`s, and the number of `a`s in ci remains equal to the sum of the number of `b`s in ci and the length of the stack. `F-INV` holds because for a computation that ends with an accept, the read `b` is the last symbol in the given word, and popping an `a` makes the stack empty, ci continues to be the read `a`s followed by the read `b`s, and, given that the stack is empty, the number of `a`s equals the number of `b`s in the consumed input. □

44.7.2 Proving L = L(P)

As before, the proof is divided into two lemmas. The first is for when w∈L and the second for when w∉L

Lemma 1 *w∈L ⇔ w∈L(P)*

Proof
(⇒) Assume w∈L. This means that w = $a^n b^n$. Given that state invariants always hold, there is a computation that has P consume all the as, then consume all the bs, and then reach F with an empty stack. Therefore, w∈L(P).

(⇐) Assume w∈L(P). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold, we may conclude that w = $a^n b^n$ and, therefore, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(P)*

Proof
(⇒) Assume w∉L. This means that w ≠ $a^n b^n$. There are three possibilities for the structure of w: w is not of the form $a^n b^m$, w is of the form $a^n b^m$, where n > m, or w is of the form $a^n b^m$, where n < m. Given that the state invariants always hold, all potential computations either fail to consume w or fail to empty the stack. In the first case, when w is not of the form $a^n b^m$, then either the first b cannot be consumed or there is an a after a b that cannot be consumed. In the second case, at least one a is left on the stack without a b to match it. In the third case, P is unable to read all the bs, because the stack becomes empty. Therefore, it is impossible for P to transition into F having read all the input with an empty stack, and we may conclude w∉L(P)

(⇐) Assume w∉L(P). This means that P cannot transition into F with an empty stack having consumed w. Given that the state invariants always hold, w must either start with a b, have an a after a b, have too many as, or have too many bs. In all cases, w∉L. □

45 A pda for L = {wcw^R | w∈(a b)*}

To further illustrate the design and implementation of pdas, we develop a machine to decide L = {wcw^R | w∈(a b)*}. As always, we follow the steps of the design recipe for state machines.

The structure of the words in the language, wcw^R, suggests how to determine if a word is in the language by reading it from left to right. The machine may read w and push it on the stack. This makes the stack w^R. After reading c, the elements of w^R may be matched with the stack elements. If all the elements after the c match all the elements on the stack, then the machine accepts.

45.1 Name and Alphabets

A descriptive name for the `pda` is `wcw^r`. The input alphabet, Σ, contains `a`, `b`, and `c`. The stack alphabet, Γ, contains `a` and `b`.

45.2 Unit Tests

The unit tests are written for words in and not in `L`. These include words that have and do have a `c`. A sample suite of unit tests is:

```
;; Tests for wcw^r
(check-equal? (sm-apply wcw^r '(a)) 'reject)
(check-equal? (sm-apply wcw^r '(a c)) 'reject)
(check-equal? (sm-apply wcw^r '(b a)) 'reject)
(check-equal? (sm-apply wcw^r '(a a b c b a b)) 'reject)
(check-equal? (sm-apply wcw^r '(c)) 'accept)
(check-equal? (sm-apply wcw^r '(a c a)) 'accept)
(check-equal? (sm-apply wcw^r '(a b b b c b b b a)) 'accept)
```

45.3 Conditions and States

To start, the consumed input, `ci`, is empty, and the stack, `s`, is empty. This condition may be captured by the staring state, `S`, and is documented as follows:

```
;;  S ci is empty and stack is empty, start state
```

From `S`, the machine moves nondeterministically to a different state, `P`, in which the machine reads `w` and pushes it onto the stack. We must carefully define what is meant by `w`. We define `w` as the (sub)word before `c`. This means that `ci` is equal to the reverse of the stack and that `c` is not in `ci`. `P` may be documented as follows:

```
;;  P ci = stack^R ∧ c not in ci
```

Observe that nondeterministically moving from `S` to `P` means that `P`'s conditions hold.

Upon reading a `c`, the `pda` may move to different state, `Q`, in which the (sub)word after the `c` is matched with the stack elements. This means that `ci` may be divided into three parts: `w` (the elements before `c`), `c`, and the elements, `v`, after `c`. Observe that `w` must equal the stack reversed appended with `v` reversed. `Q` may be documented as follows:

```
;;  Q ci = (append w (list 'c) v)  ∧
;;     w = (append stack^R v^R)
```

Observe that moving from P to Q consuming a c without modifying the stack guarantees that Q's conditions hold.

Upon reading and matching all the elements after c, the pda may move to a final state and accept. For this to occur, the stack must be empty, and the consumed input must have a (sub)word w followed by c followed by w^R. F may be documented as follows:

```
;;  F stack = '() ∧ ci = (append w (list c) w^R), final state
```

Observe that a computation that leads to accept must transition to F after matching the last elements in ci with the last element on the stack. Thus, guaranteeing that F's conditions hold.

45.4 The Transition Relation

From S, the pda only has to nondeterministically move to P. The needed transition is:

```
`((S ,EMP ,EMP) (P ,EMP))
```

In P, any symbol read from the input tape must be pushed onto the stack. When a c is read, the machine moves to Q without changing the stack. The needed transitions are:

```
`(((P a ,EMP) (P (a)))
  ((P b ,EMP) (P (b)))
  ((P c ,EMP) (Q ,EMP)))
```

In Q, any symbol read from the input tape must be matched with the symbol popped from the top of the stack. Nondeterministically, the machine decides that all the input is read, and the stack is empty to move to F. The needed transitions are:

```
`(((Q a (a)) (Q ,EMP))
  ((Q b (b)) (Q ,EMP))
  ((Q ,EMP ,EMP) (F ,EMP)))
```

There are no needed transitions from F.

45.5 Machine Implementation and Testing

Figure 64 displays wcw^R's implementation. Running the tests reveals that they all pass.

Testing with sm-test has limitations because randomly generating a word in the language is infrequent. Thus, the overwhelming majority of tests generated are for words that are correctly rejected. When faced with such a

Fig. 64 The pda implementation for L = $\{wcw^R \mid w \in (a\ b)^*\}$

```
;; L = wcw^R | w in (a b)*
;; States
;;  S ci is empty and stack is empty
;;  P ci = stack^R AND c not in ci
;;  Q ci = (append w (list 'c) v) AND
;;      w = stack^R v^R
;;  F stack = '() AND ci = (append w (list c) w^R)
(define wcw^r (make-ndpda '(S P Q F)
                          '(a b c)
                          '(a b)
                          'S
                          '(F)
                          `(((S ,EMP ,EMP) (P ,EMP))
                            ((P a ,EMP) (P (a)))
                            ((P b ,EMP) (P (b)))
                            ((P c ,EMP) (Q ,EMP))
                            ((Q a (a)) (Q ,EMP))
                            ((Q b (b)) (Q ,EMP))
                            ((Q ,EMP ,EMP) (F ,EMP)))))
;; Tests for wcw^r
(check-equal? (sm-apply wcw^r '(a)) 'reject)
(check-equal? (sm-apply wcw^r '(a c)) 'reject)
(check-equal? (sm-apply wcw^r '(b c a)) 'reject)
(check-equal? (sm-apply wcw^r '(a a b c b a b)) 'reject)
(check-equal? (sm-apply wcw^r '(c)) 'accept)
(check-equal? (sm-apply wcw^r '(a c a)) 'accept)
(check-equal? (sm-apply wcw^r '(a b b b c b b b a)) 'accept)
```

situation, it becomes important for units tests using words in the language to be thorough.

45.6 State Invariant Predicates

The invariant predicate for S must determine if both the consumed input and the stack are empty. It is implemented as follows:

```
;; word stack → Boolean
;; Purpose: Determine in the given word and stack are empty
(define (S-INV ci s) (and (empty? ci) (empty? s)))

;; Tests for S-INV
(check-equal? (S-INV '() '(a a)) #f)
(check-equal? (S-INV '(a c a) '()) #f)
(check-equal? (S-INV '(a c a) '(b b)) #f)
(check-equal? (S-INV '() '()) #t)
```

The invariant predicate for P must determine that c is not part of the consumed input and that the consumed input is equal to the stack reversed. It is implemented as follows:

```
;; word stack → Boolean
;; Purpose: Determine if the given ci is the reverse of
;;          the given stack AND c is not in ci
(define (P-INV ci s)
  (and (equal? ci (reverse s)) (not (member 'c ci))))

;; Tests for P-INV
(check-equal? (P-INV '(a c a) '(a c a)) #f)
(check-equal? (P-INV '(a a) '(a b)) #f)
(check-equal? (P-INV '() '()) #t)
(check-equal? (P-INV '(a b) '(b a)) #t)
(check-equal? (P-INV '(a b a a) '(a a b a)) #t)
```

For Q's invariant predicate, let us define v as the (sub)word after the first c and w as the (sub)word before the first c in the consumed input. The invariant predicate must determine if the consumed input is equal to the appending of w, c, and v and if w is equal to the stack reversed appended with v reversed. To achieve this, local variables are defined for w and for v. The invariant predicate is implemented as follows:

```
;; word stack → Boolean
;; Purpose: Determine if ci=s^Rv^Rcv
(define (Q-INV ci s)
  (let* [(w (takef ci (λ (s) (not (eq? s 'c)))))
         (v (if (member 'c ci)
                (drop ci (add1 (length w)))
                '()))]
    (and (equal? ci (append w (list 'c) v))
         (equal? w (append (reverse s) (reverse v))))))

;; Tests for Q-INV
(check-equal? (Q-INV '(a a) '()) #f)
(check-equal? (Q-INV '(b b c a) '(b a)) #f)
(check-equal? (Q-INV '(c) '()) #t)
(check-equal? (Q-INV '(b a c) '(a b)) #t)
(check-equal? (Q-INV '(a b c b) '(a)) #t)
(check-equal? (Q-INV '(a b b c b) '(b a)) #t)
```

Finally, the invariant predicate for F must determine if the stack is empty and if the (sub)word before c in the consumed input is equal to the reverse of the (sub)word after c. To achieve this, a local variable for the (sub)word before c in the consumed input is defined. The invariant predicate is implemented as follows:

```
;; word stack → Boolean
;; Purpose: Determine if ci=s^Rv^Rcv AND stack is empty
(define (F-INV ci s)
  (let* [(w (takef ci (λ (s) (not (eq? s 'c))))))]
    (and (empty? s)
         (equal? ci (append w (list 'c) (reverse w))))))

;; Tests for F-INV
(check-equal? (F-INV '() '()) #f)
(check-equal? (F-INV '(b b) '()) #f)
(check-equal? (F-INV '(b a c) '(b a)) #f)
(check-equal? (F-INV '(c) '()) #t)
(check-equal? (F-INV '(b a c a b) '()) #t)
(check-equal? (F-INV '(a b b c b b a) '()) #t)
```

45.7 Correctness

As before, the correctness proof is divided into two parts. The first proves that the state invariant predicates always hold. The second proves that the language of the `pda` is the language for which it was designed. We shall use the following definitions:

```
 L = wcw^R
 M = wcw^R
 w ∈ (sm-sigma M)*
 s = the stack
 F = (sm-finals M)
ci = consumed input
```

45.7.1 Proving State Invariants Hold

Theorem 2 *The state invariants hold when M accepts w.*

For the proof by induction on the number of transitions to consume `w`, we must show that `S-INV` holds when the machine starts (before consuming any input) and that invariants hold after each transition. We use the abbreviated proof notation as in the previous section.

Proof

When `M` starts, S-INV holds because ci = `'()` and s = `'()`. This establishes the base case.

Proof invariants hold after each transition that consumes input:

((S ,EMP ,EMP) (P ,EMP)): By inductive hypothesis, S-INV holds. This guarantees that ci = '() and s = '(). After using this rule, ci = '() and s = '(). P-INV holds, because c is not a member of ci and ci = (reverse s).

((P a ,EMP) (P (a))): By inductive hypothesis, P-INV holds. P-INV guarantees that ci does not contain c and that ci = (reverse s). By reading and pushing an a, ci still does not contain c, and we still have that ci = (reverse s). Thus, P-INV holds after the transition rule is used.

((P b ,EMP) (P (b))): By inductive hypothesis, P-INV holds. P-INV guarantees that ci does not contain c and that ci = (reverse s). By reading and pushing a b, ci still does not contain c, and we still have that ci = (reverse s). Thus, P-INV holds after the transition rule is used.

((P c ,EMP) (Q ,EMP)): By inductive hypothesis, P-INV holds. P-INV guarantees that ci does not contain c and that ci = (reverse s). By reading c, ci now contains c. Given that P-INV holds before the transition, after the transition, ci = (reverse s)c = (reverse s)(reverse '())c(reverse '()) = (reverse s)(reverse v)cv. That is, v = '(). Thus, Q-INV holds. After using this transition rule, M may also reach F by consuming no input on a computation that leads to accept. Given that Q-INV holds and M only moves to F if v = s = '(), we have that ci = c = '()c'() = wc(reverse w). That is, w = EMP. Thus, F-INV holds.

((Q ,EMP ,EMP) (F ,EMP)): By inductive hypothesis, Q-INV holds. Q-INV guarantees that ci = (reverse s)(reverse v)cv before using this transition. Reading nothing and not changing the stack means that ci = (reverse s)(reverse v)cv after using this transition. Recall that M is nondeterministic and uses such a transition only to move to F (the final state) and accept. This means that s must be empty and, therefore, ci = (reverse v)cv. Thus, F-INV holds.

((Q a (a)) (Q ,EMP)): By inductive hypothesis, Q-INV holds. Q-INV guarantees that ci = (reverse s)(reverse v)cv before using this transition. Reading and popping a, in essence, moves the a on the top of the stack to v. Thus, ci = (reverse s)(reverse v)cv after using this transition, and Q-INV holds. If a is the last element of the input and the stack is empty, then M moves to F. This means that ci = (reverse s)(reverse v)cv = EMP(reverse v)cv = (reverse v)cv = (reverse w)cw. That is, v = w. Therefore, F-INV holds.

((Q b (b)) (Q ,EMP)): By inductive hypothesis, Q-INV holds. Q-INV guarantees that ci = (reverse s)(reverse v)cv before using this transition. Reading and popping b, in essence, moves the b on the top of the stack to v. Thus, ci

= (reverse s)(reverse v)cv after using this transition and Q-INV holds. If `b` is the last element of the input and the stack is empty, then M moves to F. This means that ci = (reverse s)(reverse v)cv = `'()`(reverse v)cv = (reverse v)cv = (reverse w)cw. That is, v = w. Therefore, F-INV holds. □

45.7.2 Proving `L = L(M)`

As before, the proof is divided into two lemmas. The first is for when `w`∈`L` and the second for when `w`∉`L`.

Lemma 3 *w*∈*L* ⇔ *w*∈*L(M)*

Proof
(⇒) Assume w∈L. This means that w = vcv^R. Given that state invariants always hold, the following computation takes place:

`(S vcv`^{R} `EMP)` ⊢ ^∗ `(P cv`^{R} `v`^{R}`)` ⊢ `(P v`^{R} `v`^{R}`)` ⊢ ^∗ `(F EMP EMP)`

Therefore, w∈L(M).

(⇐) Assume w∈L(M). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold, we may conclude that w = vcv^R. Therefore, w∈L. □

Lemma 4 *w*∉*L* ⇔ *w*∉*L(M)*

Proof
(⇒) Assume w∉L. This means w ≠ v^{R}cv. Given that the state invariant predicates always hold, there is no computation that has M consume w and end in F with an empty stack. Therefore, w∉L(M).

(⇐) Assume w∉L(M). This means that M cannot transition into F with an empty stack having consumed w. Given that the state invariants always hold, this means that w ≠ vcv^R. Thus, w∉L. □

1 Let Σ = {`a b`}. Design and implement a `pda` for `L` = {`w` | `w` has an equal number of `a`s and `b`s}. Follow all the steps of the design recipe.

2 Let Σ = {`a b`}. Design and implement a `pda` for `L` = {`a`^{i}`b`^{j} | `i`≤`j`≤`2i`}. Follow all the steps of the design recipe.

3 Let Σ = {`a b`}. Design and implement a `pda` for `L` = {`w` | `w` is a palindrome}. Follow all the steps of the design recipe.

4 Let Σ = {`a b`}. Design and implement a `pda` for `L` = {`w` | `w` has 3 times as many `a`s than `b`s}. Follow all the steps of the design recipe.

5 Let Σ = {a b}. Design and implement a pda for L = $\{a^n b^m a^n \mid n,m \geq 0\}$. Follow all the steps of the design recipe.

6 Let Σ={a b c d}. Design and implement a pda for L=$\{a^m b^n c^p d^q : m,n,p,q \geq 0 \wedge m + n = p + q\}$. Follow all the steps of the design recipe.

7 Let Σ={a b c}. Design and implement a pda for L = $\{a^m b^n c^p : m,n,p \geq 0 \wedge (m = n \vee n = p)\}$ Follow all the steps of the design recipe.

8 Let Σ = {a b}. Design and implement a pda for L = $\{a^m b^n \mid m,n \geq 0 \wedge m \neq n\}$. Follow all the steps of the design recipe.

9 Let Σ = {a b}. Design and implement a pda for L = $\{wx \mid w,x \in \Sigma^+ \wedge w$ is a subword of $x\}$. Follow all the steps of the design recipe.

10 Let Σ = {a b}. Design and implement a pda for L = {}. Follow all the steps of the design recipe.

46 ndfas and pdas

We know that not every context-free language is regular. Recall, for example, that $a^n b^n$ is a context-free language but is not regular. In this section, we explore if for every language, L, decided by an ndfa there is a pda that decides L.

If a language, L, is regular, then there is a ndfa that decides L. Intuitively, an ndfa may be thought of as a pda that never operates on its stack. That is, it never pushes anything onto the stack, and it never pops anything off the stack. This means that given an ndfa, M, we ought to be able to build a pda, P, such that L(M) = L(P).

46.1 Design Idea

Given an ndfa, M, a pda is constructed using M's states, alphabet, starting state, and final states. The stack alphabet for the constructed pda may be '() because nothing shall every be pushed onto the stack. The pda's transition relation is built using M's transition relation. Each of M's rules is converted to a pda-rule. For an ndfa-rule, (P a R), the pda-rule ((P a EMP) (R EMP)) is generated. In this manner, the pda moves from P to R without modifying the stack just like M moves from P to R.

Fig. 65 Constructing a `pda` from an `ndfa`

```
;; ndfa → pda
;; Purpose: Convert the given ndfa to a pda
(define (ndfa->pda M)
  (let [(states (sm-states M))
        (sigma  (sm-sigma M))
        (start  (sm-start M))
        (finals (sm-finals M))
        (rules  (sm-rules M))]
    (make-ndpda states
                sigma
                '()
                start
                finals
                (map (λ (r) (list (list (first r) (second r) EMP)
                                  (list (third r) EMP)))
                     rules))))

;; Sample pda
(define ALOM-PDA  (ndfa->pda AT-LEAST-ONE-MISSING))
(define LNDFA-PDA (ndfa->pda LNDFA))

(check-equal? (sm-testequiv? ALOM-PDA AT-LEAST-ONE-MISSING) #t)
(check-equal? (sm-testequiv? LNDFA-PDA LNDFA) #t)
```

46.2 Implementation

The constructor takes as input an `ndfa` and returns a `pda` that decides the same language. It locally defines variables for the components of the given `ndfa`. The `pda` is constructed using the locally defined states, alphabet, start state, and final states unchanged. As the stack alphabet argument, the `pda` constructor is given `'()`. The list of rules is traversed using `map`. For each, a `pda`-rule is generated such that its triple contains the `ndfa`'s rule source state, consumed element, and `EMP` (to pop nothing) and its double contains the `ndfa`'s destination state and `EMP` (to push nothing).

To test the implementation of this `pda`-constructor, `LNDFA` from Fig. 24 and `AT-LEAST-ONE-MISSING` from Fig. 26 are used. The constructor is used to build a `pda` from each of these sample `ndfa`s. Test are written using `sm-testequiv?` and `pda`s created using the new constructor. An implementation of this design is displayed in Fig. 65.

11 Let `N` be an `ndfa` and let `P = (ndfa->pda N)`. Prove that `L(N) = L(P)`.

Chapter 12
Equivalence of pdas and cfgs

You probably already suspect that the set of languages accepted by pdas is the same as the set of languages generated by cfgs. That is, pdas accept any context-free language and cfgs generate any language accepted by a pda. This is stated in the following theorem:

Theorem 1 *The set of languages accepted by pdas is exactly the context-free languages.*

Our goal in this chapter is to prove the above theorem. You may notice the use of the term *accepted* instead of *decided*. As we shall discover in this chapter, some pdas do not decide a language (i.e., do not always accept or reject). Some pdas can only accept a word in the language but are unable to reject, and we shall explore why this is the case.

If the theorem above holds, then we ought to be able to build a pda for the language of a given cfg and vice versa. We break up the proof into two lemmas. The first establishes that given a cfg, G, there exists a pda, P, such that L(G) = L(P). The second establishes that given a pda, P, there exists a cfg, G, such that L(G) = L(P). For each lemma, a constructive proof is written. That is, for the first lemma, a constructor that transforms a given cfg into a pda is written. For the second lemma, a constructor that transforms a given pda into a cfg is written.

47 Building a pda from a cfg

Lemma 1 *L is a context-free language* $\Rightarrow \exists$ *pda that accepts L*

Assume L is a context-free language. This means that there is a cfg, G = (make-cfg V Σ_G R S_G), such that L=L(G).

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_12

47.1 Design Idea

We need to construct a pda, P = (make-ndpda K Σ S F Δ), such that L(P) = L(G). P must read the input from left to right. This suggests that, for a w$\in$L(G), P can simulate the leftmost derivation of w. To start, P pushes S_G onto the stack. For each cfg-rule, A $\rightarrow$ x, P pops A off the stack and pushes x onto the stack. For each a$\in\Sigma_G$, P reads a from the tape and pops a off the stack.

The design idea suggests that the pda only needs two states. The starting state, S, in which P pushes S_G and moves to, Q, the second state. P then remains in Q, which is the final state, for the rest of the computation. P may be constructed as follows:

(make-ndpda '(S Q) Σ_G $V_G \cup \Sigma_G$ 'S '(Q) Δ)

There are three types of rule in Δ:

1. ((S EMP (list S_G)) (Q EMP))
2. ((Q EMP (list 'A)) (Q (symbol->fsmlos x))), for (A $\rightarrow$ x)$\in R_G$
3. ((Q a (a)) (Q EMP)), for a$\in\Sigma_G$

The first rule type starts the leftmost derivation simulation by pushing S_G onto the stack and moving to state Q. The second rule type pops a nonterminal off the stack and pushes the right-hand side of a grammar rule by converting it to a list of valid FSM symbols. The third rule type reads an input symbol and pops a matching symbol off the stack. As mentioned above, the transition rules of P are designed to simulate the leftmost derivation of w using G. Observe that we are careful to say nothing about the behavior of P if it is ever applied to a word not in L(G).

47.2 Implementation

Based on the above design idea, the implementation of a function to convert a cfg to a pda is displayed in Fig. 66. Locally, variables for the components of the given cfg are defined. The constructed pda only has two states, S and Q, and has the same input alphabet as the given grammar. The stack alphabet are the nonterminals and terminal symbols of the given grammar, because any right-hand side of context-free grammar rule may contain elements of either. The starting state is S. The only final state is Q. The three types of rules are constructed as outlined in the design idea.

To test constructed pdas, a2nb2n from Sect. 39 and numb>numa from Sect. 40 are used. Pay close attention to the tests written. For both constructed pdas, tests are written using words in the corresponding grammars. This is certainly needed and expected because the machines are built to

Fig. 66 A function to convert a cfg to a pda

```
;; cfg → pda
;; Purpose: Transform the given cfg into a pda
(define (cfg2pda G)
  (let [(nts (grammar-nts G))
        (sigma (grammar-sigma G))
        (start (grammar-start G))
        (rules (grammar-rules G))]
    (make-ndpda '(S Q)
                sigma
                (append nts sigma)
                'S
                (list 'Q)
                (append
                 (list (list (list 'S EMP EMP) (list 'Q (list start))))
                 (map (λ (r)
                        (list (list 'Q EMP (list (first r)))
                              (list 'Q
                                    (if (eq? (third r) EMP)
                                        EMP
                                        (symbol->fsmlos (third r))))))
                      rules)
                 (map (λ (a) (list (list 'Q a (list a)) (list 'Q EMP)))
                      sigma)))))
;; Tests
(define a2nb2n-pda (cfg2pda a2nb2n))
(define numb>numa-pda (cfg2pda numb>numa))

(check-equal? (sm-apply a2nb2n-pda '(b b)) 'reject)
(check-equal? (sm-apply a2nb2n-pda '(a a b b a b)) 'reject)
(check-equal? (sm-apply a2nb2n-pda '()) 'accept)
(check-equal? (sm-apply a2nb2n-pda '(a a a b b b)) 'accept)

(check-equal? (sm-apply numb>numa-pda '(b b b)) 'accept)
(check-equal? (sm-apply numb>numa-pda '(b b a)) 'accept)
(check-equal? (sm-apply numb>numa-pda '(b b a b b)) 'accept)
(check-equal? (sm-apply numb>numa-pda '(a b b a b)) 'accept) ; ~3 minutes
```

mimic the leftmost derivation of a word. It is worth noting that pdas are computationally intensive. Thus, it may take a significant amount of time for them to accept a word. Such is the case of the last test in Fig. 66, which takes about 3 minutes to evaluate (at the time of writing). Fortunately, we are not concerned with efficiency at this moment. We are only concerned with what can be done.

Observe that tests using words not in the corresponding language are only written for a2nb2n-pda. This is because the design specifies nothing about the behavior of a pda when given a word that is not in the language. Sometimes, a constructed pda is capable of rejecting, and other times, it is incapable of rejecting. That is, a pda produced by the constructor in Fig. 66 is guaranteed to accept if the given word is in its language, but no such guarantee is made for rejection when the given word is not in its language. When a machine can

guarantee accepting any word in a given language, L, but cannot guarantee rejecting any word not in L, we say that the machine *semidecides* L. The constructor in Fig. 66 produces pdas that semidecide the language of the given cfg.

For an intuitive understanding of why a pda may only semidecide a language, it is necessary to understand how nondeterminism is implemented. The implementation of nondeterminism has the machine perform a breadth-first search for a computation to accept the given word. The search space is a tree of derivations defined by the machine's rules. A path from the root to any element of level k is a partial computation of length k. If a computation ends at level k, then breadth-first search is guaranteed to reach it. If the computation leads to accept, then the machine accepts. Otherwise, the breath-first process continues to extend the search tree to find a computation that leads to accept. If all paths in the search tree are finite and reject, then the machine rejects. What if there is a path in the search tree that is not finite? In this case, breadth-first search may continue to search for a computation that leads to an accept forever. Thus, the machine can accept but can never reject.

To understand why a2nb2n-pda decides its language and numb>numa-pda semidecides its language, we can analyze their transition relations. The transition rules for a2nb2n-pda are:

```
'(((P ϵ ϵ)   (Q (S)))
  ((Q ϵ (S)) (Q ϵ))
  ((Q ϵ (S)) (Q (a S b)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

After popping S, the machine must read input before popping another nonterminal (i.e., S) or must only read input (when there are only bs in the stack). This means that for all computations, one of two things may eventually happen. The first is that the input becomes empty, and the stack is examined to decide if the word is rejected or accepted. The second is that the machine cannot consume the whole input, which means the computation ends in reject. The machine can always say accept or reject, because all the paths in the search space are finite. Thus, we say the machine decides its language.

In contrast, the transition relation for numb>numa is:

```
'(((P ϵ ϵ) (Q (S)))
  ((Q ϵ (S)) (Q (b)))
  ((Q ϵ (S)) (Q (A b A)))
  ((Q ϵ (A)) (Q (A a A b A)))
  ((Q ϵ (A)) (Q (A b A a A)))
  ((Q ϵ (A)) (Q ϵ))
  ((Q ϵ (A)) (Q (b A)))
  ((Q a (a)) (Q ϵ))
  ((Q b (b)) (Q ϵ)))
```

Observe that after popping S, there are computations for which the stack always grows using the fourth or fifth rules without consuming any input. This means that there are computations that are infinite. A pda can only reject if all possible computations reject. It is impossible for the pda to determine this when there are potential computations that are infinite. It can accept if in Q with an empty stack and no more input to read, but it can never determine that all possible computations reject. Therefore, we say the machine semidecides its language.

47.3 Proof

Let:

```
G = (make-cfg N Σ R S)
P = (cfg2pda G)
```

To establish that `S` $^{L}\!\rightarrow_G$ `w` $\Leftrightarrow$ `(Q w S)` $\vdash^*_M$ `(Q ϵ ϵ)`, we first prove the following useful lemma:

Lemma 2
$S\ ^{L}\!\rightarrow_G \omega\alpha \Leftrightarrow (Q\ \omega\ S) \vdash^*_M (Q\ \epsilon\ \alpha)$, where $\alpha \in \{N(N \cup \Sigma)^* \cup \{\epsilon\}\} \wedge \omega \in \Sigma^*$

Let us make sure we understand what the lemma is stating. It is stating that a leftmost derivation of $\omega\alpha$ starting from S, where α is either empty or starts with the leftmost nonterminal in $\omega\alpha$, is logically equivalent to M starting in Q, reading ω, and popping S to end in Q with α on the stack. Observe that if the stack is not empty after consuming ω, then the topmost element is the next nonterminal to substitute in the leftmost derivation.

Proof ($\Rightarrow$) The proof is by induction on n = the length of the leftmost derivation of ω.

For the base case, n = 0. This means that $\omega = \epsilon$ and α = S. Thus, we have that (Q ω S) = (Q ϵ S) $\vdash^*_M$ (Q ϵ S) = (Q ϵ α).

For the inductive step:
Assume: S $^{L}\!\rightarrow_G \omega\alpha \Leftrightarrow$ (Q ω S) $\vdash^*_M$ (Q ϵ α), for n = k.
Show: S $^{L}\!\rightarrow_G \omega\alpha \Leftrightarrow$ (Q ω S) $\vdash^*_M$ (Q ϵ α), for n = k + 1.

The derivation of length k+1 looks as follows:
S $\rightarrow$ $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow u_{n+1} = \omega\alpha$
Observe that $u_n = xA\beta$, where $x \in \Sigma^* \wedge A \in N \wedge \beta \in \{N \cup \Sigma\}^*$. The derivation step from u_n to u_{n+1} substitutes A using a rule: $A \rightarrow \theta$, where $\theta \in \{N \cup \Sigma\}^*$. Therefore, $u_{n+1} = x\theta\beta$.

By inductive hypothesis, we have (Q x S) $\vdash^*$ (Q ϵ Aβ). By construction of P, there is a type 2 rule such that (Q ϵ Aβ) $\vdash$ (Q ϵ $\theta\beta$). Observe that it must be the case that the leading terminals of θ, y, are in ω and that α starts with, B, the leftmost nonterminal in θ and includes β. That is, ω = xy and α = Bβ, where y$\in\Sigma^*$ and $\beta \in$N. This means that y$\alpha = \theta\beta$. Thus, by construction of P, there are type 3 rules that consume y to yield the following computation: (Q ω S) = (Q xy S) $\vdash^*$ (Q y $\theta\beta$) = (Q y yα) $\vdash^*$ (Q ϵ α).

($\Leftarrow$) Assume (Q ω S) $\vdash^*$ (Q ϵ α). The proof is by induction on, n, the number of type 2 transitions used by P.

For the base case, n = 0. (Q ω S) $\vdash^*$ (Q ϵ α) means that $\omega = \epsilon$ and α = S. Thus, we have that S $\stackrel{L}{\rightarrow}$ $\omega\alpha = \epsilon$S = S.

For the inductive step:
Assume: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\stackrel{L}{\rightarrow}_G$ $\omega\alpha$, for n = k.
Show: (Q ω S) $\vdash^*_M$ (Q ϵ α) $\Rightarrow$ S $\stackrel{L}{\rightarrow}_G$ $\omega\alpha$, for n = k + 1.

Let ω = xy and (A $\rightarrow$ θ)$\in$R. A computation using n + 1 type 2 rules looks as follows:
(Q ω S) = (Q xy S) $\vdash^*$ (Q y Aβ) $\vdash$ (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α).
By inductive hypothesis, we have S $\stackrel{L}{\rightarrow}^*_G$ xAβ. Using the rule for A above yields S $\stackrel{L}{\rightarrow}^*_G$ x$\theta\beta$. Observe that (Q y $\theta\beta$) $\vdash^*$ (Q ϵ α) only uses type 3 rules. This means that y$\alpha = \theta\beta$. Thus, we have S $\stackrel{L}{\rightarrow}^*_G$ x$\theta\beta$ = xyα = $\omega\alpha$. □

48 Building a `cfg` from a `pda`

We now need to prove that if `P = (make-ndpda K` Σ Γ `S F` Δ`)` semidecides `L`, then `L` is a context-free language. To do so, we shall build a `cfg`, `G`, such that `L(P) = L(G)`.

48.1 Simple `pda`

It is useful to restrict the structure of `P`'s transition rules to have what we shall call a *simple* `pda`. A `pda` is simple if all transition rules have the following structure:

`((Q a` β`) (P` θ`))`, such that `Q`$\neq$`S,` $\beta \in \Gamma$, $\wedge$ $|\theta| \leq 2$

In essence, a simple pda always pops the topmost stack element and pushes zero, one, or two elements onto the stack. There are no restrictions on S, because whenever a pda starts, the stack is empty.

To transform P into a simple pda, P′, the components of P′ are defined as follows:

$$\begin{aligned}
\mathtt{K}' &= \mathtt{K} \cup \{\mathtt{X\ Y}\}, \texttt{ such that } \mathtt{X,Y} \notin \mathtt{K} \\
\Sigma' &= \Sigma \\
\Gamma' &= \Gamma \ \cup \{\mathtt{Z}\}, \text{ such that } \mathtt{Z} \notin \Gamma \\
\mathtt{S}' &= \mathtt{X} \\
\mathtt{F}' &= (\mathtt{Y}) \\
\Delta' &= \mathtt{T}(\Delta) \cup \{(\mathtt{(X\ EMP\ EMP)(S\ (Z))})\} \cup \{\forall \mathtt{W} \in \mathtt{F}\ \mathtt{((W\ EMP\ (Z))(Y\ EMP))}\}
\end{aligned}$$

The new states, X and Y, are P′'s start and only final states. The new stack symbol, Z, is the stack bottom symbol. Whenever P would have an empty stack, P′ only has Z on the stack. P′ starts a computation in state X, pushes Z onto the stack, and moves to S. Once in S, P′ simulates P's computation. P′ ends a computation that leads to accept by moving from a state in F to Y and popping Z off the stack. The function T replaces all transitions rules that violate simplicity with equivalent rules that satisfy simplicity. T replaces rules, ((Q a β) (P θ)), that violate simplicity in three steps:

1. Replace $|\beta| \geq 2$ rules
2. Replace $|\beta| = 0$ rules **without new rules with** $|\beta| \geq 2$
3. Replace $|\theta| > 2$ rules **without new rules with** $|\beta| \neq 1$

A function to convert a pda to a simple pda based on this design is displayed in Fig. 67. New symbols for the starting state, the final state, and the stack bottom are locally defined. These are used to create the initial rule and the final transition rules for the simple pda. The first step to achieve simplicity is performed by calling an auxiliary function, generate-beta<2-rules, that substitutes all rules that pop two or more elements. Given that new (intermediate) states may be needed to pop two or more elements, this auxiliary function needs as input the states of the given machine to guarantee that a repeated state is not generated. The set of rules generated is used by another auxiliary function, generate-beta=1-rules, to eliminate rules that pop nothing. To achieve this, the function needs the stack alphabet as input to create new rules. The set of rules generated all pop one element off the stack and are used by generate-theta<=2-rules to eliminate rules that push more than two elements without adding rules with $|\beta| \neq 1$. This auxiliary function may create new states to serialize pushes through several states and, thus, needs as input the set of states in the rules generated by generate-beta=1-rules. Finally, the simple pda is constructed using the generated start and final states along with the states in the rules generated by generate-theta<=2-rules, the given pda's alphabet, the generated bottom symbol and the given pda's stack alphabet, the generated start state, a list containing the generated final state, and the starting rule, the rules into the generated final state, and the rules generated by generate-theta<=2-rules.

Fig. 67 A function to convert a pda to a simple pda

```
;; pda → pda
;; Purpose: Convert given pda to a simple pda
(define (pda2spda p)
  (let*
    [(pstates (sm-states p))  (psigma (sm-sigma p))    (pgamma (sm-gamma p))
     (pstart  (sm-start p))   (pfinals (sm-finals p))  (prules (sm-rules p))
     (new-start (generate-symbol 'S pstates))
     (new-final (generate-symbol 'F pstates))
     (bottom (generate-symbol 'Z pgamma))
     (initr (mk-pda-rule new-start EMP EMP pstart (list bottom)))
     (frules (map (λ (s) (mk-pda-rule s EMP (list bottom) new-final EMP))
                  pfinals))
     (beta<2-rules (generate-beta<2-rules prules pstates))
     (beta=1-rules (generate-beta=1-rules beta<2-rules (cons bottom pgamma)))
     (theta<=2-rules (generate-theta<=2-rules beta=1-rules
                                             (extract-states beta=1-rules)))]
    (make-ndpda (append (list  new-final new-start)
                        (remove-duplicates
                         (cons pstart (extract-states theta<=2-rules))))
                psigma
                (cons bottom pgamma)
                new-start
                (list new-final)
                (cons initr (append theta<=2-rules frules)))))

;; Tests for pda2spda
(define P1 (pda2spda a2nb2n))
(define P2 (pda2spda wcwˆr))

(check-equal? (sm-testequiv? a2nb2n P1) #t)
(check-equal? (sm-testequiv? a2nb2n P1) #t)
(check-equal? (sm-testequiv? wcwˆr P2)  #t)
(check-equal? (sm-testequiv? wcwˆr P2)  #t)
```

48.1.1 Eliminating $|\beta| \geq 2$ Rules

For the first replacement step, the transition rules are partitioned based on whether or not more than one element is popped. How is `((Q a` β`) (P` θ`))`, where $|\beta = b_1 b_2 \ldots b_{\mathtt{n}}| \geq 2$, replaced? The popping of `n` elements may be done sequentially using `n` rules. The `ith` rule pops $\beta_{\mathtt{i}}$. For this `n-1` new (intermediate) states are needed to generate the needed rules. The replacement rules look as follows:

```
(((Q   EMP b1) (B1 EMP))
 ((B1  EMP b2) (B2 EMP))
 ((B2  EMP b3) (B3 EMP))
       ⋮
 ((Bn  a   bn)   (P θ)))
```

Fig. 68 Function to replace rules that pop two or more elements

```
;; (listof pda-rule) (listof state) → (listof pda-rule)
;; Purpose: Eliminate rules that pop more than two elements
(define (generate-beta<2-rules rules states)
  ;; pda-rule (listof state) → (listof pda-rule)
  ;; Purpose: Create |beta| = 1 rules for given rule
  (define (convert-beta=1 r states)
    ;; (listof symbol) (listof state) → (listof pda-rule)
    ;; Purpose: Generate pda rules for given pop list using given states
    (define (gen-intermediate-rules beta sts)
      (if (empty? (rest sts))
          '()
          (cons (mk-pda-rule (first sts) EMP (list (first beta))
                             (first (rest sts)) EMP)
                (gen-intermediate-rules (rest beta) (rest sts)))))
    (let* [(from (get-from r))  (read (get-read r))  (to (get-to r))
           (beta (get-pop r))   (push (get-push r))
           (new-states (build-list
                        (sub1 (length beta))
                        (λ (i) (generate-symbol 'B (cons 'B states)))))]
      (append
        (list
          (mk-pda-rule from EMP (list (first beta)) (first new-states) EMP)
          (mk-pda-rule (last new-states) read (list (last beta)) to push))
        (gen-intermediate-rules (rest beta) new-states))))
  (let* [(beta>=2-rules (filter (λ (r) (and (not (eq? (get-pop r) EMP))
                                            (>= (length (get-pop r)) 2)))
                                rules))
         (beta<2-rules (filter (λ (r) (not (member r beta>=2-rules))) rules))]
    (append beta<2-rules (append-map (λ (r) (convert-beta=1 r states))
                                     beta>=2-rules))))

;; Tests for generate-beta<1-rules
;; (listof pda-rule) → Boolean
;; Purpose: Determine if at most 1 element is popped by every rule
(define (all-beta<2 L)
  (andmap (λ (r) (or (eq? (get-pop r) EMP) (= (length (get-pop r)) 1))) L))

(check-pred
  all-beta<2
  (generate-beta<2-rules (list (list (list 'Q 'a '(a b c)) (list 'R '(z))))
                         '(Q R)))
(check-pred
  all-beta<2
  (generate-beta<2-rules (list (list (list 'Q 'a '(a b c)) (list 'R '(z)))
                               (list (list 'Q EMP EMP) (list 'R EMP))
                               (list (list 'Q 'b '(i j k l)) (list 'R EMP)))
                         '(Q R)))
```

The B_is are the freshly generated states. Observe that input is not read nor elements pushed until all n elements are popped.

A function based on this design idea is displayed in Fig. 68. Observe that only the rules that pop two or more elements are replaced. The auxiliary function, `convert-beta=1`, is applied to every rule that pops two or more elements. This function creates the needed new states and the first and last rules of the popping process for the given rule. The intermediate rules are generated by `gen-intermediate-rules`. This function traverses the given list of states until it is of length 1. At each step, a rule is generated to pop the next element and move to the next state.

Property-based testing is used to validate the rules generated. A predicate to test if all the generated rules pop at most a single element is implemented. Tests are written using `check-pred` and providing `generate-beta<2-rules` varying lists of `pda`-rules.

1 Implement more thorough property-based tests for `replace-beta>=2` in Fig. 68. For a given rule `r = ((P a` β`)(Q` θ`))`, for example, the first rule must start at `P`, the last rule must end at `Q`, all but the `Q` rule must read and push nothing, the `Q` rule must read `a` and push θ, and the rules must take the machine from `P` to `Q`.

48.1.2 Eliminating $|\beta| = 0$ Rules

Rule replacement step 2 is done replacing all rules `((Q a` β`) (P` θ`))`, where $|\beta| = 0$. How can this be done? Observe that popping nothing off the stack is equivalent to popping the top element and pushing it back on. Therefore, a rule that pops nothing off the stack may be replaced with rules that pop a $\gamma \in \Gamma$ and push γ back onto the stack. That is, an instance of such a rule is replaced with the following set of rules:

$\forall \gamma \in \Gamma$ `((Q a` γ`) (P` $\gamma\theta$`))`

Observe that the new rules generated do not contain rules that pop two or more elements off the stack. Therefore, the first simplicity condition is not violated by this replacement step.

To generate the needed rules, the rules that pop no elements are traversed. For each rule, the elements of Γ are traversed. For each element $\gamma \in \Gamma$, a rule is generated that pops γ and that adds γ to θ to obtain the pushed value or values. A function based on this design is displayed in Fig. 69. The function partitions the given rules into those that pop nothing and those that pop one element. The resulting list of rules contains all the rules that pop one element and the rules obtained from each rule that pops nothing. The new rules for the zero-popping rules are obtained using a `for*/list`-loop. For each zero-popping rule, the elements of the given stack alphabet (which includes the

Fig. 69 Function to substitute rules that pop nothing from the stack

```
;; (listof pda-rule) (listof symbols) → (listof pda-rules)
;; Purpose: Substitute pop nothing rules with pop 1 rules
(define (generate-beta=1-rules rls gamma)
  (let* [(beta=0-rls (filter (λ (r) (eq? (get-pop r) EMP)) rls))
         (beta>0-rls (filter (λ (r) (not (member r beta=0-rls))) rls))]
    (append beta>0-rls
            (for*/list ([r beta=0-rls]
                        [g gamma])
              (list (list (get-from r) (get-read r) (list g))
                    (list (get-to r)
                          (if (eq? (get-push r) EMP)
                              (list g)
                              (append (get-push r) (list g)))))))))

;; Tests for generate-beta=1-rules
(check-equal?
 (generate-beta=1-rules `(((P a ,EMP) (Q (a b)))) '(S a b))
 '(((P a (S)) (Q (a b S)))
   ((P a (a)) (Q (a b a)))
   ((P a (b)) (Q (a b b)))))
(check-equal?
 (generate-beta=1-rules `(((P a ,EMP) (Q (a b)))
                          ((P b (b) (Q ,EMP)))
                          ((P c ,EMP) (Q ,EMP)))
                        '(S A a b))
 '(((P b (b) (Q ϵ)))
   ((P a (S)) (Q (a b S)))
   ((P a (A)) (Q (a b A)))
   ((P a (a)) (Q (a b a)))
   ((P a (b)) (Q (a b b)))
   ((P c (S)) (Q (S)))
   ((P c (A)) (Q (A)))
   ((P c (a)) (Q (a)))
   ((P c (b)) (Q (b)))))
```

bottom of the stack symbol; see Fig. 67) are traversed. At each step, a rule that pops and pushes the current stack element is produced.

48.1.3 Eliminating $|\theta| > 2$ Rules

Rule replacement step 3 is done replacing all rules `((Q a` β`) (P` Θ`))`, where $|\Theta| > 2$. An instance of such a rule is substituted with rules that sequentially push all the elements in $\Theta =$ `'(`$\theta_1\theta_2 \ldots \theta_n$`)`. New intermediate states need to be generated for the rules that take the machine from `Q` to `P`. It is important to note that the first rule in the sequence starts at `Q` and moves the machine to the first intermediate state. The last rule in the sequence moves the machine from the last intermediate state to `P`. In order not to violate the simplicity

Fig. 70 Function to substitute rules that push more than two elements

```
;; (listof pda-rule) (listof states) → (listof pda-rule)
;; Purpose: Replace rules that push more than 2 elements
(define (generate-theta<=2-rules rls sts)
  ;; (listof pda-rule) (listof state) → (listof pda-rule)
  ;; Purpose: Generate rules with |theta|<=2 for given rules
  (define (gen-theta<=2-rules theta>2-rules sts)
    ;; pda-rule → (listof pda-rule)
    ;; Purpose: Generate |theta|<=2 rules for given rule
    (define (gen-rules r)
      ;;     (listof state) (listof symbol) (listof symbol) symbol
      ;;  → (listof pda-rule)
      ;; Purpose: Generate |theta|<=2 rules for given push and state lists
      (define (process-sts sts push pop read)
        (if (= (length sts) 2)
            (list (mk-pda-rule (first sts) read pop (second sts) push))
            (cons (mk-pda-rule (first sts) EMP pop (second sts)
                               (append pop (list (first push))))
                  (process-sts (rest sts) (rest push) pop read))))
      (let* [(from (get-from r)) (read (get-read r)) (pop (get-pop r))
             (to (get-to r))     (push (get-push r))
             (new-states (build-list
                          (sub1 (length push))
                          (λ (i) (generate-symbol 'T (cons 'T sts)))))
             (rev-push (reverse push))]
        (cons (mk-pda-rule from EMP pop (first new-states)
                           (append pop (list (first rev-push))))
              (process-sts (append new-states (list to)) (rest rev-push)
                           pop                           read))))
    (append-map gen-rules theta>2-rules))
  (let* [(theta>2-rules (filter
                         (λ (r) (and (not (eq? (second (second r)) EMP))
                                     (> (length (second (second r))) 2)))
                         rls))
         (theta<=2-rules (filter (λ (r) (not (member r theta>2-rules)))
                          rls))]
    (append theta<=2-rules (gen-theta<=2-rules theta>2-rules sts))))
;; Tests
;; (listof pda-rule) → Boolean
;; Purpose: Determine all rules have |theta|<=2
(define (all-theta<=2 rls)
  (andmap (λ (r) (<= (length (second (second r))) 2)) rls))
(check-pred
 all-theta<=2
 (generate-theta<=2-rules `(((P ,EMP (Z)) (Q (S c Z)))) '(Q P)))
(check-pred
 all-theta<=2
 (generate-theta<=2-rules `(((P ,EMP (Z)) (Q (S c Z)))
                            ((A a (Z)) (B (A b B))))
                          '(Q P)))
```

condition that β=1, each rule except the last must pop and push β. The last rule pops β, consumes a, and pushes the first element in Θ. Finally, the elements of Θ are pushed in reversed order to obtain the correct stack state. In summary, the replacement rules look like this:

$$
\begin{array}{l}
(((\mathtt{Q}\ \ \mathtt{EMP}\ \beta)\ (\mathtt{T}_1\ \ \beta\theta_n)) \\
\ ((\mathtt{T}_1\ \mathtt{EMP}\ \beta)\ (\mathtt{T}_2\ \ \beta\theta_{n-1})) \\
\ ((\mathtt{T}_2\ \mathtt{EMP}\ \beta)\ (\mathtt{T}_3\ \ \beta\theta_{n-2})) \\
\qquad \vdots \\
\ ((\mathtt{T}_n\ \mathtt{a}\ \beta)\quad (\mathtt{P}\ \theta_1)))
\end{array}
$$

Each $\mathtt{T_i}$ is a newly generated intermediate state.

A function based on this design idea is displayed in Fig. 70. The function partitions the rules in two: those that pop more than two elements and those that pop fewer elements. The result includes all the rules that pop two or less elements and the new rules generated for the rules that pop more than two elements. The generation of the new rules is done by the auxiliary function gen-theta<=2-rules that consumes the rules that pop more than two elements and the list of existing states. It appends the rules generated for each of the given rules. For each given rule, $|\theta|$-1 states are generated. The first rule of the sequence moves the machine from Q to the first intermediate state, consuming nothing and pushing $\beta\theta_n$. This rule is added to the rules generated by an auxiliary function, process-sts, which processes the rest of the intermediate states and P. This function generates rules that take the machine from the first given state to the second given state, consuming nothing, popping β, and pushing $\beta\theta_i$, where θ_i is the first of the given list of elements to push. The function stops when there are only two states left (the last newly generated state and P) generating the rule that takes the machine to P, consuming a, popping β, and only pushing θ_1 (i.e., the last element left in the given list of elements to push).

Property-based testing is used to validate the returned list of rules. The unit tests in Fig. 70 check that all rules push at most two elements.

48.1.4 Auxiliary Functions

To complete the implementation of pda2spda, all that remains is the implementation of the smaller auxiliary functions. We make the following data definition to simplify the writing of signatures:

```
A stack element, stacke, is either
  1. EMP
  2. (listof symbol)

Interpretation: The stack elements to pop or push
```

The function to make a pda-rule is:

```
;; state symbol stacke state stacke → pda-rule
;; Purpose: Build a pda-rule
(define (mk-pda-rule from a pop to push)
  (list (list from a pop) (list to push)))

;; Tests for mk-pda-rule
(define PDA-R1 (mk-pda-rule 'P 'a EMP 'Q EMP))
(define PDA-R2 (mk-pda-rule 'A 'c '(a) 'B '(b)))

(check-equal? PDA-R1 (list (list 'P 'a EMP) (list 'Q EMP)))
(check-equal? PDA-R2 (list (list 'A 'c '(a)) (list 'B '(b))))
```

The selectors for a pda-rule are:

```
;; pda-rule → state
;; Purpose: Extract from state
(define (get-from r) (first (first r)))

;; Tests for get-from
(check-equal? (get-from PDA-R1) 'P)
(check-equal? (get-from PDA-R2) 'A)

;; pda-rule → symbol
;; Purpose: Extract read symbol
(define (get-read r) (second (first r)))

;; Tests for get-read
(check-equal? (get-read PDA-R1) 'a)
(check-equal? (get-read PDA-R2) 'c)

;; pda-rule → stacke
;; Purpose: Extract pop elements
(define (get-pop r) (third (first r)))

;; Tests for get-pop
(check-equal? (get-pop PDA-R1) EMP)
(check-equal? (get-pop PDA-R2) '(a))

;; pda-rule → state
;;Purpose: Extract to state
(define (get-to r) (first (second r)))

;; Tests for get-to
(check-equal? (get-to PDA-R1) 'Q)
(check-equal? (get-to PDA-R2) 'B)
```

```
;; pda-rule → stacke
;; Purpose: Extract push elements
(define (get-push r) (second (second r)))

;; Tests for get-push
(check-equal? (get-push PDA-R1) EMP)
(check-equal? (get-push PDA-R2) '(b))
```

Finally, the function to extract the states from a given list of pda-rules is:

```
;; (listof pda-rule)→(listof state)
;; Purpose: Extract states in the given rules
(define (extract-states rls)
  (remove-duplicates
   (append-map
     (λ (r) (list (first (first r)) (first (second r))))
     rls)))

;; Tests for extract-states
(check-equal? (extract-states
                `(((P ,EMP (Z)) (Q (S c Z)))))
              '(P Q))
(check-equal? (extract-states
                `(((P ,EMP (Z)) (Q (S c Z)))
                  ((P a (Z)) (R (S c Z)))
                  ((Q ,EMP (Z)) (T (S c Z)))))
              '(P Q R T))
```

2 For an arbitrary pda, P, prove that L(P) = L((pda2spda P)).

3 Write more thorough property-based tests for generate-theta<=2-rules in Fig. 70. For a given rule r = ((P a β)(Q Θ)), for example, the first rule must start at P, the last rule must end at Q, all but the Q rule must read nothing, pop β and push two elements (the first of which must be β), and the Q rule must read a and pop β, and only push a single element.

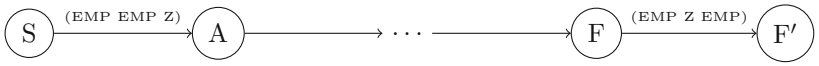
4 Carefully explain why all rules generated in replacement step 3 are guaranteed to operate on a nonempty stack.

48.2 Building a cfg from a Simple pda

The next task is to convert a simple pda to a cfg. This is done by exploiting the properties of transition rules in a simple pda. Unfortunately, the details get messy, and the algorithm is not as intuitive as the algorithm to build a pda from a cfg. On the positive side, modern applications rarely, if ever, need to build a cfg from a pda.

48.2.1 Design Idea and Data Representation

The transformation of a pda, P=(make-ndpda K Σ Γ A F Δ), to a simple pda, P′=(pda2spda P), simplifies the construction of a cfg for L(P). The simplification stems from the fact that every pda-rule pops an element and pushes at most two elements. Let us explore why this is the case. Let Z be the bottom of the stack symbol, S′ be the start state in P′, and F′ be the final state in P′. P′ is outlined as follows:



The labels on the arrows, from left to right, contain a read, a pop, and a push element. The middle part simulates P. The grammar that is constructed shall simulate P′. It needs to generate all words that take P′ from A to F′ by popping Z. We may represent these words as a triple: (S Z F′). This means that the starting nonterminal for the constructed grammar, S, must generate (S Z F′):

S → (A Z F′)

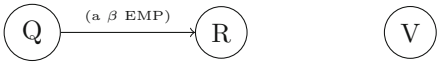
The representation of nonterminals as triples is an implementation choice. To build an actual cfg, of course, these triples need to be converted to symbols. In general, we shall talk about all words that take P′ from some state Q to some state R by popping θ and represent them as (Q θ R). All such triples represent the nonterminals of the grammar that is constructed and are formally defined as follows:

```
;; An list nonterminal, lnt, is a (list state symbol state).

;; Interpretation:
;;   All words that take the pda from the first state to
;;   the second state by popping the symbol off the stack.
```

Given that an lnt represents a nonterminal, how can a cfg-rule be represented? Clearly, lnts need to be used. The left-hand side of a rule may be represented as a symbol (as the rule above) or as an lnt. We define a cfg-rule's left-hand side as follows:

```
;; A lhs is either:
;;  1. symbol
;;  2. lnt

;; Interpretation:
;;  The left-hand side of a cfg-rule is represented as either a
;;  a symbol or an lnt.
```

How can the right-hand side of a cfg-rule be represented? To answer this, consider the rules in a simple pda. Every rule has a symbol for what is read and a symbol for what is popped. The simple pda rules vary in what is pushed. A rule pushes 0, 1, or two elements. That is, there are three types of rules in a simple pda. Let us first consider rules that push 0 elements: `((Q a` β`) (R EMP))`. This rule takes the machine from `Q` to `R` by popping nothing. After using this rule, the machine may transition in zero or more steps to an arbitrary state `V`. Graphically, we may describe the use of such a rule as follows:

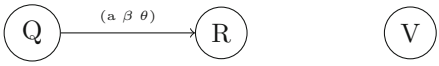


The grammar needs to generate all words that take `P′` from `Q` to `V` by popping β and pushing nothing. Such words are generated by `(Q` β `V)`. To get to `V`, the machine reads `a` to reach `R` and then pops nothing to reach `V` in zero or more moves.[11] Thus, the needed rule is:

`(Q` β `V)` $\rightarrow$ `a(R EMP V)`

Let `NS` be `P′`'s states except `(sm-start P′)`. $\forall$V$\in$`NS` such a rule is generated. Such a rule states that the words that take the machine from `Q` to `V` by popping β may start with `a` and be followed by a word that takes the machine from `R` to `V` by pushing nothing. Thus, we have that the right-hand side of a cfg-rule may be represented by a symbol followed by an `lnt`.

Now consider the cfg-rule needed when a single element is pushed:



The grammar needs to generate all words that take `P′` from `Q` to `V` by popping β and pushing θ. This situation is similar to pushing nothing except that after reaching `R` θ must be popped. The needed cfg-rule is:

`(Q` β `V)` $\rightarrow$ `a(R` θ `V)`

Once again, such a rule is generated for all states `V` in `P′` except `(sm-start P′)`. It is opportune to note that the above rule is not stating that all (sub)computations from `R` to `V` pop θ. It only refers to (sub)computations from `R` to `V` that pop θ. That is, there may be (sub)computations that pop

[11] The rules that take the machine from `R` to `V` may, of course, make use of the stack.

something different, and such cfg-rules are generated from different pda-rules. That observed, we have that such a cfg-rule right-hand side may also be represented by a symbol followed by an lnt.

Consider the pda-rules that push two elements: ((Q a β) (R $\theta\tau$)). To reach V from Q, the two elements must be pushed and popped. After popping the first element, the machine may be in an arbitrary state W. Graphically, consider the following:

The grammar must be able to generate a word that starts with a followed by a word that takes P′ from R to W by popping θ and ending with a word that takes P′ from W to V by popping τ. Thus, the needed cfg-rule may be represented as follows:

(Q β V) $\rightarrow$ a(R θ W)(W τ V)

Such a rule is generated for all states V in P′ except P′'s start state. Observe that the right-hand side of a cfg-rule may be represented by a symbol followed by two lnts.

Finally, we must consider the stopping conditions for word generation by the grammar. The three types of production rules generated above always generate at least one lnt. Once all the terminal symbols are generated, how are the remaining nonterminals eliminated? Observe that if P′ is in an arbitrary state Q, then it may remain in Q without popping or consuming any input. Therefore, the following rules are needed:

$\forall$Q$\in$(sm-states P′) (Q EMP Q) $\rightarrow$ EMP

Note that the right-hand side of a cfg-rule may also be represented using a symbol.

The representation of the right-hand side of a cfg-rule has variety and may be defined as follows:

```
;; A rhs is either:
;;  1. symbol
;;  2. (list symbol lnt)
;;  3. (list symbol lnt lnt)
;; Interpretation:
;;  A cfg-rule right hand side is represented as either a
;;  symbol or a list with a symbol and either one or two lnt.
```

We can now define the representation of a cfg-rule as follows:

```
;; A cfg-rl is a (list lhs ARROW rhs)
;; Interpretation:
;;   A cfg-rl represents a cfg-fule as a list with an lhs,
;;   an ARROW, and a rhs.
```

Finally, to simplify code development, the following function is defined to construct a cfg-rl:

```
;; lhs rhs → cfg-rl
;; Purpose: Create a cfg-rl
(define (mk-cfg-rl a-lhs a-rhs) (list a-lhs ARROW a-rhs))

;; Tests for mk-cfg-rl
(check-equal? (mk-cfg-rl 'S 'aSb) (list 'S ARROW 'aSb))
(check-equal? (mk-cfg-rl '(S Z F) '(a (P Z Q)))
              (list '(S Z F) ARROW '(a (P Z Q))))
```

48.2.2 Implementation

Figure 71 displays the body of an implementation for a function, pda2cfg, to convert a pda to a cfg. It is tested by converting a^nb^n from Fig. 63 and wcw^r from Fig. 64 into cfgs. The function's body is a let*-expression that defines variables for the simple pda obtained from the given pda, the components of the simple pda, a new starting nonterminal, and a new bottom of the stack symbol. The starting cfg-rl is constructed using the generated starting nonterminal for the left-hand side and the given pda's start state, the bottom symbol, and the simple pda's only final state. To generate the other cfg-rls, the simple pda's set of states, not including the start state, and the simple pda's rules, not including rules from the start state, are defined. The production rules for when 0, 1, or 2 elements are pushed are independently computed by traversing the appropriate rules and the states not including the start state. The rules to stop word generation are constructed by traversing the simple pda's states. To actually build a cfg, a symbol must be generated for each lnt in the generated production rules. An association table is created whose entries contain an lnt and the corresponding nonterminal symbol generated for it. The table is used to generate the needed cfg-rules (i.e., rules with no lnts). Finally, the grammar is constructed by calling make-cfg with the generated starting nonterminal and the states in the generated table, the simple pda's alphabet, the generated production rules, and the generated starting nonterminal.

To generate the production rules for pda-rules that push nothing, the given rules are traversed. For each rule, the given states are traversed. The needed traversals are performed using a for*/list-loop. The lhs is constructed using the current rule's from-state, the current rule's only pop element, and the current state. The rhs is constructed using the current's rule read element and an lnt constructed using the current rule's to-state, EMP, and the current state. The function is implemented as follows:

Fig. 71 The function to convert a pda to a cfg

```
;; pda → cfg     Purpose: Convert given pda to a cfg
(define (pda2cfg P)
       ⋮
  (let* [(p (pda2spda P))
         (pstates (sm-states p)) (psigma (sm-sigma p))   (pgamma (sm-gamma p))
         (pstart (sm-start p))   (pfinals (sm-finals p)) (prules (sm-rules p))
         (start (generate-symbol 'S '(S)))
         (bottom (first (filter (λ (s) (not (member s (sm-gamma P))))
                                pgamma)))
         (startr (mk-cfg-rl start (list (sm-start P) bottom (first pfinals))))
         (pstates-nostart (remove pstart pstates))
         (prules-nostartrls (filter (λ (r)
                                      (not (eq? (first (first r)) pstart)))
                                    prules))
         (theta=0-prs (gen-theta=0-prs (filter (λ (r) (eq? (get-push r) EMP))
                                               prules-nostartrls)
                                       pstates-nostart))
         (theta=1-prs (gen-theta=1-prs (filter (λ (r) (keep-rule? r 1))
                                               prules-nostartrls)
                                       pstates-nostart))
         (theta=2-prs (gen-theta=2-prs (filter (λ (r) (keep-rule? r 2))
                                               prules-nostartrls)
                                       pstates-nostart))
         (self-prs (map (λ (s) (mk-cfg-rl (list s EMP s)  EMP))
                        pstates))
         (st-list (cons (third startr)
                        (extract-lnts theta=0-prs theta=1-prs
                                      theta=2-prs self-prs)))
         (st-tbl (map (λ (lnt) (list lnt (generate-symbol 'G '(G)))) st-list))
         (new-rls (make-cfg-rules startr      st-tbl      theta=0-prs
                                  theta=1-prs theta=2-prs self-prs))]
    (make-cfg (cons start (map second st-tbl)) psigma new-rls start)))
;; Tests for pda2cfg
(define a2nb2n-grammar (pda2cfg a^nb^n))
(define wcw^r-grammar (pda2cfg wcw^r))

(check-equal? (last (grammar-derive a2nb2n-grammar '(a b))) 'ab)
(check-equal? (last (grammar-derive a2nb2n-grammar '(a a b b))) 'aabb)
(check-equal? (last (grammar-derive wcw^r-grammar '(a c a))) 'aca)
(check-equal? (last (grammar-derive wcw^r-grammar '(a b c b a))) 'abcba)
```

```
;; (listof pda-rule) (listof state) → (listof cfg-rl)
;; Purpose: Return cfg-rls for given |theta|=0 pda rules
(define (gen-theta=0-prs rls sts)
  (for*/list ([r rls]
              [s sts])
    (mk-cfg-rl (list (get-from r) (first (get-pop r)) s)
               (list (get-read r) (list (get-to r) EMP s)))))
```

To generate the production rules for pda-rules that push one element, the given rules are traversed. For each rule, the given states are traversed. The needed traversals are performed using a for*/list-loop. The lhs is constructed using the current rule's from-state, the current rule's only pop element, and the current state. The rhs is constructed using the current's rule read element and an lnt constructed using the current rule's to-state, the current rule's only push element, and the current state. The function is implemented as follows:

```
;; (listof pda-rule) (listof state) → (listof cfg-rl)
;; Purpose: Return cfg-rls for given |theta|=1 pda rules
(define (gen-theta=1-prs rls sts)
  (for*/list ([r rls]
              [s sts])
    (mk-cfg-rl (list (get-from r) (first (get-pop r)) s)
               (list (get-read r)
                     (list (get-to r)
                     (first (get-push r)) s)))))
```

To generate the production rules for pda-rules that push two elements, the given rules are traversed. For each rule, the given states are traversed. Let us call each of state in this traversal s1. For each s1, the given states are traversed. Let us call each of state in this second state traversal s2. The needed traversals are performed using a for*/list-loop. The lhs is constructed using the current rule's from-state, the current rule's only pop element, and s2. The rhs is constructed using the current's rule read element and two lnts. The first lnt is constructed using the current rule's to-state, the current rule's first push element, and s1. The second lnt is constructed using s1, the current rule's second push element, and s2. The function is implemented as follows:

```
;; (listof pda-rule) (listof state) → (listof cfg-rl)
;; Purpose: Return cfg-rls for the given |theta|=2 pda rules
(define (gen-theta=2-prs rls sts)
 (for*/list ([r rls] [s1 sts] [s2 sts])
  (mk-cfg-rl (list (get-from r) (first (get-pop r)) s2)
             (list (get-read r)
                   (list (get-to r) (first (get-push r)) s1)
                   (list s1 (second (get-push r)) s2)))))
```

To transform the generated cfg-rls to cfg-rules, each set of cfg-rl types is first traversed to substitute lnts with their corresponding nonterminals and to eliminate leading EMPs from rhss. For each cfg-rl to stop the word generation, a cfg-rule is constructed by using the nonterminal for lhs extracted from the given table and the rule's right-hand side (i.e., EMP). For each cfg-rl that pops no elements or pops a single element, a cfg-rule is constructed by using the nonterminal for lhs extracted from the given table. The rule's

right-hand side is constructed by examining the rhs's first element. If it is not EMP, a new rhs is constructed by converting the first element and the nonterminal for the rhs' lnt (extracted from the given table) to a symbol. If it is EMP, then a new rhs is constructed using only the nonterminal for the rhs' lnt (extracted from the given table). For each cfg-rl that pops two elements, a cfg-rule is constructed by using the nonterminal for lhs extracted from the given table. The rule's right-hand side is constructed by examining the rhs's first element. If it is not EMP, a new rhs is constructed using the first element and the two nonterminals for the rhs' lnts extracted from the given table. If the first element is EMP, then a new rhs is constructed using only the two nonterminals for the rhs' lnts extracted from the given table. The function is written as follows:

```
;;    cfg-rl          (listof (list lnt nt)) (listof cfg-rl)
;;    (listof cfg-rl) (listof cfg-rl)        (listof cfg-rl)
;;  →
;;    (listof cfg-rule)
;; Purpose: Convert all cfg-rls to cfg-rules using given
;;          association table
(define (make-cfg-rules startr      st-tbl      theta=0-prs
                        theta=1-prs theta=2-prs self-prs)
 (cons (mk-cfg-rl (first startr)
                  (get-nt (third startr) st-tbl))
       (append
        (map (λ (rl)
               (mk-cfg-rl (get-nt (first rl) st-tbl)
                          (third rl)))
             self-prs)
        (map (λ (rl)
               (mk-cfg-rl
                (get-nt (first rl) st-tbl)
                (if (not (eq? (first (third rl)) EMP))
                    (los->symbol (list (first (third rl))
                                       (get-nt (second (third rl))
                                               st-tbl)))
                    (get-nt (second (third rl)) st-tbl))))
             (append theta=0-prs theta=1-prs))
        (map (λ (rl)
               (mk-cfg-rl
                (get-nt (first rl) st-tbl)
```

```
              (if (not (eq? (first (third rl)) EMP))
                  (los->symbol (list (first (third rl))
                                     (get-nt (second (third rl))
                                             st-tbl)
                                     (get-nt (third (third rl))
                                             st-tbl)))
                  (los->symbol (list
                                (get-nt (second (third rl))
                                        st-tbl)
                                (get-nt (third (third rl))
                                        st-tbl))))))
          theta=2-prs))))
```

The extraction of the lnts from the cfg-rls is done by appending the lnts found in each cfg-rl type and removing duplicates. For rules that pop 0 or 1 element, the returned lnts are the lhs and the second element of the rhs. For rules that pop 2 elements, the returned lnts are the lhs and the second and third elements of the rhs. Finally, for rules that stop word generation only the lhs is returned. The function is implemented as follows:

```
;;    (listof cfg-rl) (listof cfg-rl)
;;    (listof cfg-rl) (listof cfg-rl)
;;  →
;;    (listof lnt)
;; Purpose: Extract the lnts in the given cfg-rl
(define (extract-lnts theta=0-prs theta=1-prs
                      theta=2-prs self-prs)
  (remove-duplicates
   (append
    (append-map (λ (pr) (list (first pr)
                              (second (third pr))))
                (append theta=0-prs theta=1-prs))
    (append-map (λ (pr) (list (first pr)
                              (second (third pr))
                              (third (third pr))))
                theta=2-prs)
    (map first self-prs))))
```

The remaining auxiliary local functions determine if the rhs of a given cfg-rl is of a given length and extract from a given association table the nonterminal associated with a given lnt. These functions are implemented as follows:

```
;; cfg-rl natnum → Boolean
;; Purpose: Determine if rule has a rhs of given length
(define (keep-rule? r len)
  (and (not (eq? (get-push r) EMP))
       (= (length (get-push r)) len)))

;; lnt (listof (list lnt nt)) → nt
;; Purpose: Extract nonterminal for given lnt from given table
(define (get-nt a-lnt tbl) (second (assoc a-lnt tbl)))
```

5 For an arbitrary `pda`, `P`, let `G = (pda2cfg P)`, prove that `L(P) = L(G)`. The proof may be done by induction on the length of a derivation. Start by proving the following claim:

$$\texttt{(Q A P)} \rightarrow_G^* \texttt{x} \Leftrightarrow \texttt{(Q x A)} \vdash^* \texttt{(P EMP EMP)}$$

6 In a simple `pda`, the only transition from the starting state is `((S′ EMP EMP) (A (Z)))`. Carefully explain why the starting production rule in the generated grammar is `S → (A Z F′)` and not `S′ → (S EMP F′)`.

7 The grammar returned by `(pda2cfg P)` may contain useless nonterminals. A useless nonterminal is one that can never be generated in any derivation. Design and implement a function to remove useless nonterminals from a grammar returned by `(pda2cfg P)`.

Chapter 13
Properties of Context-Free Languages

Chapter 12 established that we have two different ways of demonstrating that a language, L, is context-free. We can either build a cfg for L or we can build a pda for L. Which ought we use? The answer is that we ought to use whichever is easier for us. Sometimes, it is easier to design and implement a pda, and other times, designing and implementing a cfg is easier. This flexibility is afforded to because we know that for every context-free L, there is a cfg that generates its members and a pda that accepts its members.

In this chapter, we explore new ways to establish that a language, L, is context-free. Specifically, we shall study closure properties of context-free languages. These properties are very much in the spirit of the closure properties for regular languages studied in Sect. 30. That said, the number of operations under which context-free languages are closed is fewer. In addition, we shall explore how to prove that a language is not context-free. This is done by proving a pumping theorem for context-free languages much like we did for regular languages.

49 Union

Context-free languages are closed under union. Given two context-free languages, $\mathtt{L}_1$ and $\mathtt{L}_2$, we shall build a cfg for $\mathtt{L}_1 \cup \mathtt{L}_2$.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_13

49.1 Design Idea

If L_1 and L_2 are context-free languages, then there are cfgs, G_1, and G_2, such that L_1 = L(G_1) and L_2 = L(G_2). Without loss of generality, we assume that (grammar-nts G_1) ∩ (grammar-nts G_2) = ∅. The grammar for $L_1 \cup L_2$ nondeterministically decides to simulate a derivation using G_1 or G_2.

A cfg for $L_1 \cup L_2$ is built using the following components:

V = (grammar-nts G_1) ∪ (grammar-nts G_2) ∪ {A}
Σ = (grammar-sigma G_1) ∪ (grammar-sigma G_2)
R = (grammar-rules G_1) ∪ (grammar-rules G_2) ∪
{(S ARROW (grammar-start G_1)) (S ARROW (grammar-start G_2))}
S = A = (generate-symbol 'S (append (grammar-nts G_1) (grammar-nts G_2)))

A is a freshly generated nonterminal symbol for the starting nonterminal, such that A ∉ (grammar-nts G) ∪ (grammar-nts H). The set of nonterminal symbols contains A, G_1's nonterminals and G_2's nonterminals. The alphabet contains G_1's and G_2's alphabets. The set of rules has G_1's and G_2's rules. In addition, the set of rules has a transition from A to G_1's starting nonterminal and from A to G_2's starting nonterminal. These rules give the grammar the choice to use G_1 or G_2 for a derivation.

49.2 Implementation

The cfg constructor takes as input two cfgs, G1 and G2, and returns a cfg. In order to guarantee that the given grammars do not share any nonterminals the nonterminals in G2 are renamed using FSM's grammar-rename-nts. The starting symbol for the constructed grammar, A, is generated, making sure it is not a nonterminal of either G1 or the renamed G2. The cfg constructor is called with a list that contains A and the nonterminals in G1 and the renamed G2, a list containing G1's and in renamed G2's alphabets without duplicates, a list with the two new rules from A and all the rules in G1 and renamed G2, and A. An implementation of the constructor is displayed in Fig. 72.

The constructor is tested using a2nb2n from Sect. 39, numb>numa from Sect. 40, and MULT3-as from Fig. 56. Grammars are constructed for the union of a2nb2n and numb>numa and for union of numb>numa and MULT3-as. For each of these constructed grammars, words not in the union language as well as words in the language of one input grammar but not the other are tested.

Fig. 72 A constructor for the union of two `cfg`s

```
;; cfg cfg → cfg
;; Purpose: Construct a grammar for the union of the given grammars
(define (cfg-union G1 G2)
  (let* [(H (grammar-rename-nts (grammar-nts G1) G2))
         (G-nts (grammar-nts G1))     (H-nts (grammar-nts H))
         (G-sigma (grammar-sigma G1)) (H-sigma (grammar-sigma H))
         (G-rules (grammar-rules G1)) (H-rules (grammar-rules H))
         (G-start (grammar-start G1)) (H-start (grammar-start H))
         (A (generate-symbol 'S (append G-nts H-nts)))]
    (make-cfg (cons A (append G-nts H-nts))
              (remove-duplicates (append G-sigma H-sigma))
              (append (list (list A ARROW G-start)
                            (list A ARROW H-start))
                      G-rules
                      H-rules)
              A)))

(define a2nb2nUnumb>numa (cfg-union a2nb2n numb>numa))
(define numb>numaUMULT3-as (cfg-union numb>numa MULT3-as))

(check-equal? (grammar-derive a2nb2nUnumb>numa '(b a))
              "(b a) is not in L(G).")
(check-equal? (grammar-derive a2nb2nUnumb>numa '(b a a a b b))
              "(b a a a b b) is not in L(G).")
(check-equal? (last (grammar-derive a2nb2nUnumb>numa '(a b)))
              'ab)
(check-equal? (last (grammar-derive a2nb2nUnumb>numa '(a a b b)))
              'aabb)
(check-equal? (last (grammar-derive a2nb2nUnumb>numa '(b b)))
              'bb)
(check-equal? (last (grammar-derive a2nb2nUnumb>numa '(b a a b b)))
              'baabb)

(check-equal? (grammar-derive numb>numaUMULT3-as '(a a a a b b))
              "(a a a a b b) is not in L(G).")
(check-equal? (grammar-derive numb>numaUMULT3-as '(a a))
              "(a a) is not in L(G).")
(check-equal? (last (grammar-derive numb>numaUMULT3-as '(b b b)))
              'bbb)
(check-equal? (last (grammar-derive numb>numaUMULT3-as '(a b a b a b)))
              'ababab)
```

49.3 Proof

Let `G1` and `G2` be `cfg`s, let `L` = `L(G1)`∪`L(G2)`, and let `G = (cfg-union G1 G2)`.

Theorem 1 *Context-free languages are closed under union.*

Proof
Assume S $\rightarrow_G$ w. By construction of `G`, the derivation of `w` starts with `(grammar-start G)`, and then either `(grammar-start G1)` is generated or `(grammar-start G2)` is generated. If `(grammar-start G1)` is generated, then `G` simulates `G1`. If `(grammar-start G2)` is generated, then `G` simulates `G2`. Recall that `(grammar-nts G1)`$\cap$`(grammar-nts G2)` $= \varnothing$. This means that either `G1` generates `w` or `G2` generates `w`. Thus, `w` $\in$ `L(G1)`$\cup$`L(G2)`. □

1 Why does the proof that context-free languages are closed under union require that `(grammar-nts G1)`$\cap$`(grammar-nts G2)` $= \varnothing$. What can go wrong if `(grammar-nts G1)`$\cap$`(grammar-nts G2)` $\neq \varnothing$?

50 Concatenation

Context-free languages are closed under concatenation. Given two context-free languages, L_1 and L_2, we shall build a `cfg` for L_1L_2, the language of all words that start with a word in L_1 followed by a word in L_2.

50.1 Design Idea

There are `cfg`s, G_1 and G_2, such that L_1 `= L(`G_1`)` and L_2 `= L(`G_2`)`. Without loss of generality, we assume that `(grammar-nts` G_1`)` $\cap$ `(grammar-nts` G_2`)` `=` $\varnothing$. We shall build a grammar, `G`, for L_1L_2.

Every word in L_1L_2 may be written as `w = xy`, such that `x` $\in$ L_1 and `y` $\in$ L_2. A `cfg` for L_1L_2 is built using the following components:

V = `(grammar-nts` G_1`)` $\cup$ `(grammar-nts` G_2`)` $\cup$ {`A`}
Σ = `(grammar-sigma` G_1`)` $\cup$ `(grammar-sigma` G_2`)`
R = `(grammar-rules` G_1`)` $\cup$ `(grammar-rules` G_2`)` $\cup$
{`S ARROW (grammar-start` G_1`)(grammar-start` G_2`)`}
S = `A = (generate-symbol 'S (append (grammar-nts` G_1`) (grammar-nts` G_2`)))`

`A` is a freshly generated starting nonterminal for the grammar constructed such that `A` $\notin$ `(grammar-nts` G_1`)` $\cup$ `(grammar-nts` G_2`)`. The constructed grammar starts by generating `(grammar-start` G_1`)(grammar-start` G_2`)`. G_1 is simulated by `G` to derive `x` from `(grammar-start` G_1`)`, and G_2 is simulated by `G` to derive `y` from `(grammar-start` G_2`)`.

Fig. 73 A constructor for the concatenation of two context-free languages

```
;; cfg cfg → cfg
;; Purpose: Build a cfg for the concatenation of the given cfgs
(define (cfg-concat G1 G2)
  (let* [(G2 (grammar-rename-nts (grammar-nts G1) G2))
         (G1-nts   (grammar-nts G1))   (G2-nts (grammar-nts G2))
         (G1-sigma (grammar-sigma G1)) (G2-sigma (grammar-sigma G2))
         (G1-rules (grammar-rules G1)) (G2-rules (grammar-rules G2))
         (G1-start (grammar-start G1)) (G2-start (grammar-start G2))
         (A (generate-symbol 'S (append G1-nts G2-nts)))]
    (make-cfg (cons A (append G1-nts G2-nts))
              (remove-duplicates (append G1-sigma G2-sigma))
              (append
                (list (list A ARROW (los->symbol (list G1-start G2-start))))
                G1-rules
                G2-rules)
              A)))

;; Tests for cfg-concat
(define a2nb2n-numb>numa (cfg-concat a2nb2n numb>numa))
(define MULT3-as-a2nb2n (cfg-concat MULT3-as a2nb2n))

(check-equal? (grammar-derive a2nb2n-numb>numa '(b a))
              "(b a) is not in L(G).")
(check-equal? (grammar-derive a2nb2n-numb>numa '(a a a))
              "(a a a) is not in L(G).")
(check-equal? (grammar-derive a2nb2n-numb>numa '(a a b b))
              "(a a b b) is not in L(G).")
(check-equal? (last (grammar-derive a2nb2n-numb>numa '(b b b))) 'bbb)
(check-equal? (last (grammar-derive a2nb2n-numb>numa '(a b b))) 'abb)
(check-equal? (last (grammar-derive a2nb2n-numb>numa '(a a b b b b b)))
              'aabbbbb)

(check-equal? (grammar-derive MULT3-as-a2nb2n '(a a))
              "(a a) is not in L(G).")
(check-equal? (grammar-derive MULT3-as-a2nb2n '(b b b b a))
              "(b b b b a) is not in L(G).")
(check-equal? (last (grammar-derive MULT3-as-a2nb2n '(a a b b))) 'aabb)
(check-equal? (last (grammar-derive MULT3-as-a2nb2n '(b b b b b b))) 'bbbbbb)
```

50.2 Implementation

Based on our design idea, Fig. 73 displays the implementation of a `cfg` constructor that takes as input, `G1` and `G2`, two `cfg`s. As done by `cfg-union`, the nonterminals of one of the given grammars are renamed to guarantee that the intersection of nonterminals is empty. A fresh symbol for the constructed grammar's starting nonterminal is generated. To construct the grammar, the fresh starting nonterminal is added to the nonterminals of the given grammars, and a new starting rule that generates the concatenation of the given grammars' starting nonterminals is added to the rules of the given grammars.

50.3 Proof

Theorem 2 *Context-free languages are closed under concatenation.*

It is straightforward to see that `L((cfg-concat G1 G2))` is a `cfg` for `L(G1)L(G2)`. The proof is left as an exercise.

2 Prove Theorem 2.

3 Why is it necessary to rename `G2`'s nonterminals in Fig. 73? What can go wrong if `(grammar-nts G1)`$\cap$`(grammar-nts G2)` $\neq \varnothing$?

51 Kleene Star

Context-free languages are closed under Kleene star. Given a context-free language, `L`, we shall build a `cfg` for `L`*, the language of all words obtained by concatenating 0 or more words in `L`.

51.1 Design Idea

Given that `L` is a context-free language, there exists a `G` = `(make-cfg N` Σ `R S)` such that `L` = `L(G)`. A `cfg` for `L`* must generate 0 or more words in `L`. Each word in `L` is generated by `S`. Thus, a `cfg` for `L`* must generate 0 or more `S`s.

A `cfg` for `L`* may be constructed using the following components:

V = `(grammar-nts G)` $\cup$ {`A`}
Σ = `(grammar-sigma G)`
R = `(grammar-rules G)` $\cup$ {`(S ARROW EMP) (S ARROW SS`}
S = `A = (generate-symbol 'S (grammar-nts G))`

A fresh new symbol, `A`, is generated for the starting nonterminal. The set of nonterminals is defined by adding `A` to `G`'s nonterminals. The alphabet is `G`'s alphabet. To obtain the set of rules, two new rules, to generate an arbitrary number of `G`'s starting nonterminal, are added to `G`'s rules.

51.2 Implementation

The implementation of the `cfg` constructor is left as an exercise.

4 Implement, `(cfg-ks G)`, a `cfg` constructor for $\texttt{L}^*$.

51.3 Proof

Theorem 3 *Context-free languages are closed under Kleene star.*

The proof of the theorem is left as an exercise.

5 Prove Theorem 3 by showing that `L* = L((cfg-ks G))`, where `L` is an arbitrary context-free language such that `L = L(G)`.

52 The Pumping Theorem for Context-Free Languages

Like regular languages, there is an infinite number of context-free languages. Also like regular languages, an infinite context-free language exhibits periodicity to create longer and longer words. For example, a derivation using a `cfg` may repeatedly cycle through a set of nonterminals generating the same terminal symbols to obtain a longer word in the language.

There are, however, languages in the universe that are not context-free. In this section, we explore how to prove that a language is not context-free. We shall prove a pumping theorem for context-free languages that may be used to prove that a language is not context-free. This, of course, means that any such language is also not regular.

52.1 Yield Length

Before proceeding with the theorem, it is useful to bound the length of a parse tree's yield. That is, we wish to describe a parse tree's yield's length in terms related to the parse tree and to the grammar. Given a grammar `G`, we define κ as the longest right-hand side of any rule. For instance, consider the following `cfg`s:

```
;; L = a^nb^n
(define a2nb2n (make-cfg '(S)
                         '(a b)
                         `((S ,ARROW ,EMP)
                           (S ,ARROW aSb))
                         'S))

;; L = w | w in (a b)* AND  w has more b than a
(define numb>numa (make-cfg '(S A)
                            '(a b)
                            `((S ,ARROW b)
                              (S ,ARROW AbA)
                              (A ,ARROW AaAbA)
                              (A ,ARROW AbAaA)
                              (A ,ARROW ,EMP)
                              (A ,ARROW bA))
                            'S))
```

For `a2nb2n`, κ is 3 because its longest right-hand side is `aSb`. For `numb>numa`, κ is 5.

Given a `cfg G`, the yield's length for any parse tree generated by `G` is bounded by the height of the tree and κ. Let `T` be a parse tree generated by `G` of height `h`.

Lemma 1 *T's yield's length* $\leq \kappa^h$.

Proof

The proof is by induction on h.

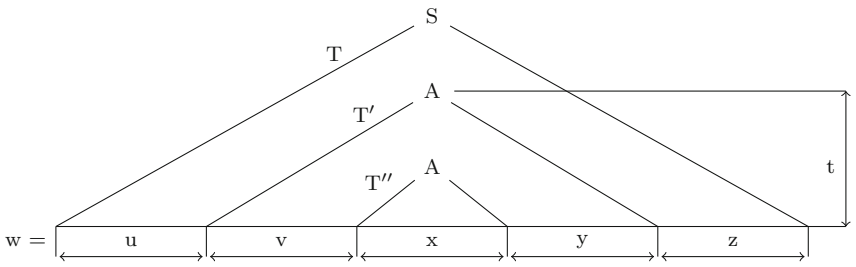
Base case: h = 1
A parse tree of height 1 means that a single rule has been used. This means that the yield's length may be at most κ. Thus, yield's length $\leq \kappa = \kappa^1 = \kappa^h$.

Inductive Step
Assume: T's yield's length $\leq \kappa^n$, for h = n
Show: T's yield's length $\leq \kappa^{n+1}$, for h = n + 1

Consider the structure of a parse tree of height n + 1. It has a root and at most κ subtrees of height n. Let T_i be any of these subtrees. By inductive hypothesis, T_i's yield's length $\leq \kappa^n$. If every T_i's yield's length is κ^n then T's yield's length is at most $\kappa * \kappa^n$. Thus, T's yield's length $\leq \kappa^{n+1}$. □

Let `w`$\in$`L(G)` such that $|$`w`$| > \kappa^h$. Observe that Lemma 1 informs us that `w`'s parse tree must have a path longer than `h`.

Fig. 74 Parse tree for w with a repeated nonterminal



52.2 The Pumping Theorem

The following definitions are assumed to state and prove the theorem:

```
G = (make-cfg V Σ R S)    v = |V|    w ∈ L(G)
```

Theorem 4 *If* $|w| > \kappa^{|V|}$ *then* $w = uvxyz$ *such that* $(v \neq \varepsilon \vee y \neq \varepsilon)$, $\exists k\ |vxy| \leq k$, *and* $\forall n \geq 0\ uv^n xy^n z \in L(G)$.

The theorem states that if `w` is long enough (i.e., $|w| > \kappa^{|V|}$), then it may be divided into five subwords (i.e., `u`, `v`, `x`, `y`, and `z`) such that `v` or `y` are not empty, the length of `vxy` is at most `k` (a constant derived from the parse tree), and any word obtained by repeating or suppressing both `v` and `y` the same number of times is in `L(G)`. The ability to repeat `v` and `y` an arbitrary number of times captures the periodicity mentioned above and makes it clear that `G` must generate an infinite language.

Proof

Let T be the parse tree for w rooted at S that has the smallest number of leaves among all possible parse trees for w. Given that $|w| > \kappa^{|V|}$, there is a path in T (from S to a leaf) of length at least |V| + 1. This path has |V| + 2 nodes of which the last must be a terminal symbol (i.e., a member of w). Thus, such a path must have |V| + 1 nonterminals. This means that there must be a repeated nonterminal on this path.

The parse tree for w may be visualized as displayed in Fig. 74. The parse tree, T, is rooted at S and generates w. During the derivation S generates uAz. That is, S generates the beginning, u, and the end, z, of w with the nonterminal A in between. The parse tree rooted at this A is T′ and A generates vAy. Observe that A is a repeated nonterminal. The parse tree rooted at the second A is T″, and it generates x.

Observe that the part of T′ excluding T″ may be repeated an arbitrary number of times or may be excluded resulting in a word that is in L(G). That is, $uv^n xy^n z \in L(G)$ for $n \geq 0$.

Let t be the height of T′ (as indicated in Fig. 74). Observe that the yield of T′, vxy, has a length of at most κ^t. This is the constant k that bounds the length of the subword that contains the parts that may be repeated an arbitrary number of times: $|\text{vxy}| \leq \kappa^t = k$.

Finally, observe that if vy = EMP, then there is a parse tree with fewer leaves that generates w. Specifically, the following derivation is captured by a parse tree with fewer leaves:

$$\texttt{S} \rightarrow^* \texttt{uAz} \rightarrow^* \texttt{uxz} = \texttt{w}$$

That is, the smaller parse tree omits the derivation of A from A. This contradicts our assumption that T is the parse tree for w with the smallest number of leaves among all possible parse trees for w. Therefore, vy ≠ EMP. This means that $0 < |vxy| \leq k \leq \kappa^{|V|}$. □

52.3 Applying the Pumping Theorem for Context-Free Languages

The pumping theorem is useful to prove that a language, L, is not context-free. We abbreviate κ^v as K. As with the use of the pumping theorem for regular languages, think of the use of the pumping theorem for context-free languages as a game against an opponent that will try to demonstrate that the conditions of the theorem can be satisfied. You pick a word that is long enough. That is, choose a word w such that $|\text{w}| \geq \text{K}$. Your opponent searches for values for u, v, x, y, and z such that $\text{vy} \neq \text{EMP}$, $|\text{vxy}| \leq K$, and $\forall \text{n} \geq 0$ $\text{uv}^n\text{xy}^n\text{z} \in \text{L}$. If no such values exists, then you may conclude that L is not context-free.

Claim $\text{L} = \{\text{a}^\text{n}\text{b}^\text{n}\text{c}^\text{n} |\ \text{n} \geq 0\}$ is not context-free.

Proof

Let $\text{w} = \text{a}^k\text{b}^k\text{c}^k$. We must determine the possible values vxy may take. We can observe that vxy may not contain as, bs, and cs because the length of vxy must be less than or equal to k. Observe that vy may contain only one letter variety or may contain two letter varieties by straddling the border between as and bs or between bs and cs. We prove that L is not context-free as follows:

vy	Argument
a^+	Pump up once on v and y. The resulting word has more as than bs and than cs. Therefore, it is not in L.
b^+	Pump up once on v and y. The resulting word has more bs than as and than cs. Therefore, it is not in L.
c^+	Pump up once on v and y. The resulting word has more cs than as and than bs. Therefore, it is not in L.
a^+b^+	Pump up once on v and y. The resulting word has more as and b than cs. Therefore, it is not in L.
b^+c^+	Pump up once on v and y. The resulting word has more bs and c than as. Therefore, it is not in L.

No matter where the vxy window is placed in w, a word that is not in w may be generated. Therefore, we may conclude that L is not context-free. □

Claim L = {a^n | n is a square} is not context-free.

Proof

Let w = a^{k^2}. Clearly, w is in L. Given that w only contains as then vy = a^i, where $i > 0$. If we pump up once the resulting word is a^{K^2+i}. Observe that there are no words in L with a length greater than $|a^{K^2}|$ and less than $|a^{(K+1)^2}|$. That is, after a^{K^2} the next word in L has length $K^2 + 2K + 1$. In order for a^{K^2+i} to be in L, i must at a minimum be of length $2K + 1$. This, however, is impossible because |vxy| must be less than or equal to k. Thus, i cannot be greater than or equal to $2k + 1$.

No matter where the vxy window is placed in w, a word that is not in w may be generated. Therefore, we may conclude that L is not context-free. □

6 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that L = {www | w ∈ Σ^*} is not context-free.

7 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that L = {w | w has equal number of as, bs, and cs} is not context-free.

8 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that L = {$a^m b^n c^p$ | n ≤ minimum(m, p)} is not context-free.

9 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that L = {wxw^R | x,w ∈ Σ^* ∧ $|w| = |x|$} is not context-free.

10 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that $L = \{ww \mid w \in \Sigma^*\}$ is not context-free.

11 Let Σ = {a b}. Use the pumping theorem for context-free languages to show that L = {w | w is a palindrome with equal number of as and bs} is not context-free.

53 Context-Free Languages Are Not Closed Under Intersection nor Complement

Unlike regular languages, context-free languages are not closed under intersection nor complement. The following theorems establish these facts.

Theorem 5 *Context-free languages are not closed under intersection.*

Proof

Let $L_1 = \{a^n b^n c^m \mid m,n \geq 0\}$ and $L_2 = \{a^m b^n c^n \mid m,n \geq 0\}$. Observe that both of these languages are context-free. Consider $L = L_1 \cap L_2$. Clearly, $L = \{a^n b^n c^n \mid n \geq 0\}$, which we know is not context-free. Therefore, context-free languages are not closed under intersection. □

Theorem 6 *Context-free languages are not closed under complement.*

Proof

$L_1 \cap L_2$ = (complement ((complement L_1) $\cup$ (complement L_2))). If context-free languages were closed under complement, then they would also be closed under intersection, which we know they are not. Therefore, context-free languages are not closed under complement. □

54 Intersection of a Context-Free Language and a Regular Language

Although context-free languages are not closed under intersection, there exist context-free languages that result in a context-free language when intersected. Recall that all regular languages are context-free. We claim that the intersection of a context-free language and a regular language is a context-free language. How can we prove this? As you may have already guessed, we ought to be able to develop a constructive proof and, therefore, a new constructor for a context-free language.

54.1 Design Idea

If L_1 is a context-free language and L_2 is a regular language, then there is a pda, M_1 = (make-ndpda K_1 Σ_1 Γ S_1 F_1 R_1), and a dfa, M_2 = (make-dfa K_2 Σ_2 S_2 F_2 R_2), such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. How can we conclude that $L_1 \cap L_2$ is a context-free language? We shall build a pda that simultaneously simulates M_1 and M_2. This new pda shall only accept if both M_1 and M_2 accept.

The basic idea is to have each state of the new pda be a *super state* that represents a state in M_1 and a state in M_2. The new pda shall simulate all the transitions of M_1. In addition, it will also track the transitions made by M_2. If M_1 has a rule that moves from P to Q on an a, then for every state $Y \in M_2$, the new pda has a rule that moves from super state (P Y) to super state (Q T), such that (Y a T) $\in$ R_2. If M_1 has a rule that moves from P to Q on EMP, then for every state $Y \in M_2$, the new pda has a rule that moves from super state (P Y) to super state (Q Y) to simulate that M_2 does not change state.

We define push/pop elements, a super state, and pda super rule as follows:

```
;; Push or pop elements, stacke, is either
;;  1. EMP
;;  2. (listof symbol)

;; A super state, ss, is a (list state state)

;; A pda super rule, ssrule, is:
;;  (list (list ss symbol stacke) (list ss stacke))
```

The first state in an ss is a state in M_1, and the second state is a state in M_2. The super states for the new pda are obtained by computing the Cartesian product of M_1's and M_2's states. The new pda's input and stack alphabets are those of M_1. The constructed pda's starting ss represents M_1's and M_2's starting states. The final sss for the constructed pda are given by the Cartesian product of M_1's and M_2's final states. Finally, to build the pda, a state must be generated for each sss and substituted into the set of states, the starting state, the final states, and the rules of the constructed pda.

54.2 Implementation

Based on our design idea, an implementation of the constructor is outlined in Fig. 75. The body of the function is a let*-expression that locally defines the set of super states, the starting super state, and the set of final super states. An auxiliary function is used to compute the Cartesian product of two (listof state). The rules of the given pda are partitioned into those that do not read from the input tape and those that do. Each of these sets

Fig. 75 Constructor for context-free and regular language intersection

```
;; pda dfa → pda  Purpose: Build intersection pda from given machines
(define (pda-intersect-dfa a-pda a-dfa)
  ;; (listof state) (listof state) → (listof ss)
  ;; Purpose: Create a list of super states
  (define (cartesian-product pda-sts dfa-sts) ...)

  ;; (listof pda-rule) (listof state) → (listof ssrule)
  ;; Purpose: Create ssrules for given empty transition pda-rules
  (define (make-EMP-rls EMP-pda-rls dfa-sts) ...)

  ;; (listof pda-rule) (listof state) (listof dfa-rls) → (listof ssrule)
  ;; Purpose: Create ssrules for given nonempty transition pda-rules
  (define (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls) ...)

  ;; (listof ss) → (listof (list ss state))
  ;; Purpose: Create table associating given super states with a new state
  (define (make-ss-table K) ...)

  ;; (listof ssrule) (ss → state) → (listof pda-rule)
  ;; Purpose: Convert ssrules to pda-rules
  (define (convert-ssrules ss-rls ss->state) ...)

 (let* [(pda-sts (sm-states a-pda))  (dfa-sts (sm-states a-dfa))
        (pda-rls (sm-rules a-pda))   (dfa-rls (sm-rules a-dfa))
        (K (cartesian-product pda-sts dfa-sts))
        (start (list (sm-start a-pda) (sm-start a-dfa)))
        (F (cartesian-product (sm-finals a-pda) (sm-finals a-dfa)))
        (non-EMP-pda-rls (filter
                           (λ (r) (not (eq? (second (first r)) EMP))) pda-rls))
        (EMP-pda-rls (filter (λ (r) (eq? (second (first r)) EMP)) pda-rls))
        (non-EMP-rls (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls))
        (EMP-rls (make-EMP-rls EMP-pda-rls dfa-sts))
        (ss-tbl (make-ss-table K))
        (ss->state (λ (ss) (second (assoc ss ss-tbl))))]
   (make-ndpda (map ss->state K)  (sm-sigma a-pda)  (sm-gamma a-pda)
               (ss->state start)  (map ss->state F)
               (convert-ssrules (append non-EMP-rls EMP-rls) ss->state))))
```

are converted to `ss-rule`s using an auxiliary function. A table is created that associates each `ss` with a freshly generated state by calling an auxiliary function. This table is used to define a function, `ss->state`, that consumes a `ss` and returns its associated freshly generated state. Finally, the `pda` is constructed by using this function to substitute `ss`s with their associated state.

The Cartesian product of two `(listof state)` is computed using a `for*/list`. For every state in the first given list, the states in the second given list are traversed. Each super state in constructed by creating a list with the current state from the first list and the current state from the second list. The function is implemented as follows:

```
;; (listof state) (listof state) → (listof ss)
;; Purpose: Create a list of super states
(define (cartesian-product pda-sts dfa-sts)
  (for*/list [(s1 pda-sts) (s2 dfa-sts)] (list s1 s2)))
```

The `ssrule`s for the given `pda`'s transitions that consume no input are constructed using the states from the `dfa`. A `for*/list` loop is used to traverse the given states for each of the given rules. For the current rule, `r`, and the current state, `s`, an `ssrule` is constructed. The triple of this rule contains the `ss` built from the from-state in `r` and `s`, `EMP`, and the `stacke` popped by `r`. The double of the rule contains the `ss` built from the to-state in `r` and the `stacke` pushed by `r`. The function is implemented as follows:

```
(define (make-EMP-rls EMP-pda-rls dfa-sts)
  (for*/list [(r EMP-pda-rls) (s dfa-sts)]
    (list (list (list (first (first r)) s)
                EMP
                (third (first r)))
          (list (list (first (second r)) s)
                (second (second r))))))
```

The `ssrule`s for the given `pda`'s transitions that consume input are constructed using the states and the rules from the `dfa`. The construction process is the same as the ones used for the `pda`'s transitions that consume no input with two exceptions. Instead of `EMP`, the element consumed by the current `pda`-rule, `i`, is used. Instead of using `s`, the current `dfa` state to build the destination super state, the destination super state is built using the state the `dfa` transitions to from `s` on `i`. The function is implemented as follows:

```
(define (make-nonEMP-rls non-EMP-pda-rls dfa-sts dfa-rls)
  (for*/list [(r non-EMP-pda-rls) (s dfa-sts)]
    (list (list
            (list (first (first r)) s)
            (second (first r))
            (third (first r)))
          (list
            (list
              (first (second r))
              (third
              (first
                (filter (λ (rl)
                          (and (eq? (first rl) s)
                               (eq? (second rl)
                                    (second (first r)))))
                        dfa-rls))))
            (second (second r))))))
```

The `ss`-table is constructed by traversing the generated super states. For each super state, a fresh state is generated using `FSM`'s `generate-symbol`. The `ssrule`s are converted by traversing them. For each, its super states are converted to a state using `ss->state`. The functions are implemented as follows:

```
(define (make-ss-table K)
 (for*/list [(ss K)] (list ss (generate-symbol 'T '(T)))))

(define (convert-ssrules ss-rls ss->state)
 (for*/list [(r ss-rls)]
  (list
   (list (ss->state (first (first r)))
         (second (first r))
         (third (first r)))
   (list (ss->state (first (second r)))
         (second (second r))))))
```

Finally, tests are written using `M` (the `dfa` from Sect. 21.1), `NO-ABAA` (the `dfa` from Fig. 18), and `a^nb^n` (the `pda` from Fig. 63). A `pda` is defined for the intersection of `a^nb^n` with each of the `dfa`s. Words that are rejected and accepted in each of the intersection languages are used to write tests as follows:

```
;; Tests for pda-intersect-dfa
(define a^nb^nIM (pda-intersect-dfa a^nb^n M))
(define a^nb^nINO-ABBA (pda-intersect-dfa a^nb^n NO-ABAA))

(check-equal? (sm-apply a^nb^nIM '()) 'reject)
(check-equal? (sm-apply a^nb^nIM '(a b b)) 'reject)
(check-equal? (sm-apply a^nb^nIM '(a a b b)) 'reject)
(check-equal? (sm-apply a^nb^nIM '(a b)) 'accept)

(check-equal? (sm-apply a^nb^nINO-ABBA '(b b)) 'reject)
(check-equal? (sm-apply a^nb^nINO-ABBA '(a a b)) 'reject)
(check-equal? (sm-apply a^nb^nINO-ABBA '()) 'accept)
(check-equal? (sm-apply a^nb^nINO-ABBA '(a a b b)) 'accept)
```

54.3 Proof

Theorem 7 *The intersection of a context-free language and a regular language is context-free.*

Proof (Sketch)
Let Q be a `pda`, let D be a `dfa`, and let M = (pda-intersect-dfa Q D). By

construction, M simulates the transitions of both Q and D. M starts its simulations in a state that represents Q's and D's starting states and only accepts if both Q and D accept. This means that all accepted strings must be in L(Q) and L(D). Thus, L(M) = L(Q) ∩ L(D). □

The previous theorem may be used to prove a language, L, is context-free. To do so, we must show that L is obtained from the intersection of a context-free language and a regular language.

Claim L = { w∈{a b}*| w does not contain abaa and has equal number of as and bs.} is context-free.

Proof The language of all words that do not contain abaa is decided by NO-ABAA from Fig. 18. Thus, L(NO-ABAA) is regular. The language, E, of all words that have an equal number of as and bs is generated by the following context-free grammar:

S → EMP
  → aSb
  → bSa
  → aSbSbSa
  → bSaSaSb

Thus, E is context-free. Observe that L = L(E) ∩ L(NO-ABAA). By Theorem 7, we may conclude that L is context-free. □

12 Let P_1 and P_2 be pdas. Show how to construct a pda for $L(P_1) \cup L(P_2)$, thus providing an alternate proof for Theorem 1. Make sure to implement and test your construction algorithm.

13 Let P_1 and P_2 be pdas. Show how to construct a pda for $L(P_1)L(P_2)$, thus providing an alternate proof for Theorem 2. Make sure to implement and test your construction algorithm.

14 Let P_1 be a pda. Show how to construct a pda for $L(P_1)^*$, thus providing an alternate proof for Theorem 3. Make sure to implement and test your construction algorithm.

15 Let C be a context-free language and let R be a regular language. Consider the language, L(C - R), of all words in C that are not in R. Is L(C - R) context-free? If so, implement a constructor for L(C - R). Otherwise, explain why L(C - R) may not be context-free.

16 Let C be a context-free language and let R be a regular language. Consider the language, L(R - C), of all words in R that are not in C. Is L(R - C) context-free? If so, implement a constructor for L(R - C). Otherwise, explain why L(R - C) may not be context-free.

Chapter 14
Deterministic PDAs

In this chapter we explore deterministic pdas in more detail. As for a dfa, a deterministic pda's transition relation is a function. That is, from a given configuration, C_i, there is at most one transition that is applicable. That is, the pda never has a choice of what to do next. For this to be the case the pda's transitions must satisfy the following constraints:

1. There are no pair of transitions that offer a choice on which to use
2. For all accepting states, Q, there are no transitions of this nature: ((Q EMP EMP) (P a)). That is, the machine does not have to choose between continuing with the computation or accepting.

Observe that the second constraint means that any transition out of a final state must consume input or pop an element off the stack.

In our study of finite-state automata, we discovered that dfas and ndfas are equivalent. That is, given an ndfa we can construct a dfa that decides the same language. The natural question that arises is: Given a nondeterministic pda, can a deterministic pda that decides the same language be constructed? If this is possible then, much like for finite-state machines, we are free to design a pda using nondeterminism with the assurance that it can transformed into a deterministic pda and implemented in any programming language beyond FSM.

55 A Deterministic pda for wcw^R

To start, we shall become familiar with designing a deterministic pda. The process is almost the same as designing a nondeterministic pda. The difference, of course, is that the pda must have a transition function. To explore such a design we first revisit designing a machine to decide $L = wcw^R$.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_14

55.1 Design Idea

The nondeterministic pda's operation for L developed in Sect. 45 may be summarized as follows:

1. Nondeterministically move from S, the starting state, to P
2. In P, accumulate w on the stack in reversed order and move to Q upon reading a c without modifying the stack.
3. In Q, match the elements of w^R with the elements on the stack and nondeterministically move to, F, the final state.

A deterministic pda for L must avoid nondeterministic transitions. Observe that the nondeterministic pda operates in two phases. Before reading c, it pushes w onto the stack. After reading c, it matches w^R with the elements on the stack. The phases are clearly delineated around reading c. This means a deterministic pda may decide L as follows:

1. Before reading c, push read elements onto the stack
2. After reading c, match read elements with elements onto the stack

55.2 Name, Alphabets, and Unit Tests

A descriptive name is dwcw^r. The input alphabet is {a b c} and the stack alphabet is {a b}.

The units tests illustrate words that are and that are not in L. Sample tests are:

```
(check-equal? (sm-apply dwcw^r '())) 'reject)
(check-equal? (sm-apply dwcw^r '(a b b a)) 'reject)
(check-equal? (sm-apply dwcw^r '(a a)) 'reject)
(check-equal? (sm-apply dwcw^r '(a b b c b b a b))
              'reject)
(check-equal? (sm-apply dwcw^r '(a c a)) 'accept)
(check-equal? (sm-apply dwcw^r '(b a c a b)) 'accept)
(check-equal? (sm-apply dwcw^r '(b b a c a b b)) 'accept)
```

55.3 Condition, States, Transition Function, and Implementation

The design idea suggests the need for two states: one for each phase around reading a c. In the first state, S, the elements before c are pushed onto the stack. This state is documented as follows:

S: ci = {a b}* AND stack = ci^R, starting state

Fig. 76 A deterministic pda for wcw^R

```
#lang fsm
;; L = wcw^R
;; State Documentation
;;  S: ci = a b* AND stack = ci^R
;;  F: ci = xycy^R AND stack = x^R
(define dwcw^r (make-ndpda '(S F)
                           '(a b c)
                           '(a b)
                           'S
                           '(F)
                           `(((S a ,EMP) (S (a)))
                             ((S b ,EMP) (S (b)))
                             ((S c ,EMP) (F ,EMP))
                             ((F a (a)) (F ,EMP))
                             ((F b (b)) (F ,EMP)))))

(check-equal? (sm-apply dwcw^r '()) 'reject)
(check-equal? (sm-apply dwcw^r '(a b b a)) 'reject)
(check-equal? (sm-apply dwcw^r '(a a)) 'reject)
(check-equal? (sm-apply dwcw^r '(a b b c b b a b)) 'reject)
(check-equal? (sm-apply dwcw^r '(a c a)) 'accept)
(check-equal? (sm-apply dwcw^r '(b a c a b)) 'accept)
(check-equal? (sm-apply dwcw^r '(b b a c a b b)) 'accept)
```

In the second state, F, a c has been read and each read element after c has been matched. This means that we may think of w as equal to xy, where x is the unmatched part and y is the matched part. Under this light, the consumed input is wcy^R and the stack is x^R. This state is documented as follows:

F: ci = $xycy^R$ AND stack = x^R, final state

In S, the machine pushes as and bs onto the stack without changing states. If a c is read the machine transitions to F to begin the second phase. The needed transition rules are:

```
((S a ,EMP) (S (a)))
((S b ,EMP) (S (b)))
((S c ,EMP) (F ,EMP))
```

In F, the machine matches each read a or b with the element on the top of the stack and does not change state. The needed transitions are:

```
((F a (a)) (F ,EMP))
((F b (b)) (F ,EMP))
```

Observe that there are no nondeterministic choices that the machine must make using the rules above. Therefore, if the machine is correct, we have that L is a deterministic context-free language. The implementation of the machine is displayed in Fig. 76.

55.4 State Invariant Predicates

The invariant predicate for S determines if the consumed input only contains as and bs and that the consumed input and the stack reversed are equal. It is implemented as follows:

```
;; word stack → Boolean
(define (S-INV ci s)
  (and (andmap (λ (s) (or (eq? s 'a) (eq? s 'b))) ci)
       (equal? ci (reverse s))))

(check-equal? (S-INV '(c) '()) #f)
(check-equal? (S-INV '(b a) '(b a)) #f)
(check-equal? (S-INV '() '()) #t)
(check-equal? (S-INV '(a b b) '(b b a)) #t)
```

The invariant predicate for F determines that a c has been read, that the consumed input is of the form $xycy^R$ and that the stack equals x^R. A local variable may be defined for the subword before the c and from it x and y are extracted. The predicate invariant is implemented as follows:

```
;; word stack → Boolean
(define (F-INV ci s)
  (and (member 'c ci)
       (let* [(w (takef ci (λ (x) (not (eq? x 'c)))))
              (x (take ci (length s)))
              (y^R (drop ci (add1 (length w))))]
         (and (equal?
                ci
                (append x (reverse y^R) (list 'c) y^R))
              (equal? s (reverse x))))))

(check-equal? (F-INV '(a b c) '()) #f)
(check-equal? (F-INV '(a a c) '(a)) #f)
(check-equal? (F-INV '(a b) '(b a)) #f)
(check-equal? (F-INV '(c) '()) #t)
(check-equal? (F-INV '(b a c) '(a b)) #t)
(check-equal? (F-INV '(a b b c b) '(b a)) #t)
```

55.5 Correctness

We shall use the following definitions:

```
L = wcw^R        M = dwcw^r          v ∈ (sm-sigma M)*
s = the stack    ci = consumed input
```

55.5.1 Proving State Invariants Hold

Theorem 1 *The state invariants hold when M accepts v.*

For the proof by induction, on the number of transitions to consume v, we must show that S-INV holds when the machine starts (before consuming any input) and that invariants hold after each transition.

Proof

For the base case, when M starts ci = '() and s = '(). This means that ci only contains as and bs and that ci equals s reversed. Thus, S-INV holds.

Proof invariants hold after each transition that consumes input:

((S a ,EMP) (S (a))): By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.

((S b ,EMP) (S (b))): By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, the consumed input only contains as and bs and the consumed input is equal to the stack reversed. Thus, S-INV holds.

((S c ,EMP) (F ,EMP)): By inductive hypothesis S-INV holds. This means that ci$\in${a b}* and that ci = s^R. After using this rule, ci=xycy^R, where x$\in${a b}* and y='(), and stack=x^R. Thus, F-INV holds.

((F a (a)) (F ,EMP)): By inductive hypothesis F-INV holds. This means that ci=xycy^R, where x,y$\in${a b}*, and stack=x^R. Using this rule means that x's last element is an a. That is, x=x'a. After using this rule, ci=x'aycy^{R}a= x'ayc(ay)^R and stack=x'^{R}. Thus, F-INV holds.

((F b (b)) (F ,EMP)): By inductive hypothesis F-INV holds. This means that ci=xycy^R, where x,y$\in${a b}*, and stack=x^R. Using this rule means that x's last element is a b. That is, x=x'b. After using this rule, ci=x'bycy^{R}b= x'byc(by)^R and stack=x'^{R}. Thus, F-INV holds. □

55.5.2 Proving L = L(M)

Lemma 1 *w$\in$L $\Leftrightarrow$ w$\in$L(M)*

Proof

($\Rightarrow$) Assume w$\in$L. This means that w = vcv^R. Given that state invariants always hold, there is a computation that has M consume w and reach F with an empty stack:

$$(\mathtt{S\ vcv}^R\ \mathtt{EMP}) \vdash^* (\mathtt{S\ cv}^R\ \mathtt{v}^R) \vdash (\mathtt{F\ v}^R\ \mathtt{v}^R) \vdash^* (\mathtt{F\ EMP\ EMP})$$

Therefore, w∈L(M).

(⇐) Assume w∈L(M). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that w = vcv^R. Therefore, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(M)*

Proof
(⇒) Assume w∉L. This means w ≠ v^{R}cv. Given that the state invariant predicates always hold, there is no computation that has M consume w and end in F with an empty stack. Therefore, w∉L(M).

(⇐) Assume w∉L(M). This means that M does not halt in F with an empty stack having consumed w. Given that the state invariants always hold, this means that w ≠ vcv^R. Thus, w∉L. □

56 A Deterministic pda for L = {a^{m}b^{n}c^p| m≠n ∧ m,n,p>0}

Designing a deterministic pda for wcw^R is relatively straightforward because the machine's phases are well delineated around the reading of c. Not every deterministic pda may have well delineated phases. Consider, for example, the following language:

$$\mathtt{L} = \{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^p \mid m \neq n \wedge m,n,p > 0\}$$

Every word in L either has more as than bs or vice versa. It is not immediately clear, however, how the machine may deterministically decide that excess as or excess bs need to be processed.

56.1 Design Idea

Intuitively, the machine ought to push all the as onto the stack, match bs, either pop excess as or read excess bs, and read cs. The challenge is to deterministically determine if excess as need to be popped or excess bs need to be read. If there is an excess of bs in the input word then all the as on the stack will be matched and the stack becomes empty. This is a problem because the machine cannot nondeterministically decide to keep matching as on the stack or read excess bs. If the machine could sense that the stack is

empty then the decision becomes deterministic. If the stack is empty read excess bs. Otherwise, match as as bs are read. Another potential problem is deterministically transitioning to the final state. To prevent nondeterministic transitions of this nature, a special end of input symbol, z, not in the input alphabet of the machine, shall be used. All input for this deterministic pda shall be wz, where w is the word the machine must decide to accept or reject.

How can the machine sense that the stack is empty? The machine cannot sense when the stack is empty. Instead, we shall endow the machine with the ability to detect that there is an excess of bs. We shall use a technique developed to construct simple pdas (see Sect. 48.1): the machine shall use a bottom of the stack symbol. The first element pushed onto the stack by this machine is the bottom of the stack symbol and the last element popped off the stack before accepting is also the bottom of the stack symbol. This element is used to detect when the stack would be empty if not used. For our purposes, y serves as the bottom of the stack symbol.

The machine's phases may be outlined as follows:

1. When started, if an a is read then (a y) is pushed onto the stack.
2. All the remaining as read are pushed onto the stack.
3. Match a read b by popping an a
4. Decide if there are more as or more bs:
    a. If a c is read and there is an a on the stack then there are more as than bs. The machine transitions to a state to read the remaining cs and pop the remaining as. Upon reading z, the machine pops y and moves to the final state.
    b. If a b is read and y is at the top of the stack then there are morebs than as. The machine transitions to read the bs with changing the stack. It then transitions to read the cs without altering the stack. Upon reading z, the machine pops y and moves to the final state

56.2 Name, Alphabets, and Unit Tests

A descriptive machine name is ambncp. The input alphabet is '(a b c z), where z, as mentioned above, is only used to mark the end of the input. The stack alphabet is '(a y), where y is, as mentioned above, the bottom of the stack symbol.

To test the machine, words that do not have z at the end, that have an equal number of as and bs, that have letters out of order, and that fail to have an a, a b, or a c are tested to illustrate that they are rejected. Words that have a z at the end, no letters out of order, at least one a, one b, and one c and either more as than bs or vice versa are tested to illustrate that they are accepted. Sample tests are:

```
(check-equal? (sm-apply ambncp '(z)) 'reject)
(check-equal? (sm-apply ambncp '(a b b c c)) 'reject)
(check-equal? (sm-apply ambncp '(a a b b c c z)) 'reject)
(check-equal? (sm-apply ambncp '(a b a z)) 'reject)
(check-equal? (sm-apply ambncp '(a b b z)) 'reject)
(check-equal? (sm-apply ambncp '(a a a b b c c z))
              'accept)
(check-equal? (sm-apply ambncp '(a a a b b b b c c c z))
              'accept)
```

56.3 Conditions, States, Transition Function, and Implementation

According to the design idea, the machine starts with an empty stack and having consumed no input. The state for this condition is documented as follows:

```
S: ci = stack = '(), starting state
```

Upon reading an a, the machine transitions to A to push the rest of the as read. The only transition needed out of S is:

```
((S a ,EMP) (A (a y)))
```

In A, the machine pushes the read as onto the stack. This state, A, is documented as follows:

$$\texttt{A: ci = } (\texttt{a}^{+}) \wedge \texttt{ stack = } (\texttt{a}^{+}\ \texttt{y})$$

Upon reading a b, the machine pops an a and transitions to a state, B, to match read bs with as on the stack. Observe that an a may be popped because at least one a must be read to reach A. The needed transitions are:

```
((A a ,EMP) (A (a)))
((A b (a))  (B ,EMP))
```

In B, read bs are matched with as on the stack. This means the consumed input has 1 or more as followed by 1 or more bs, the number of as is less than or equal to the number of bs, and the stack only contains unmatched as and y. This state is documented as follows:

$$\texttt{B: ci = } (\texttt{a}^{\texttt{i}}\ \texttt{b}^{\texttt{j}}) \wedge \texttt{ stack = } (\texttt{a}^{\texttt{i-j}}\ \texttt{y}) \wedge \texttt{ 0 < j <= i}$$

In this state, the decision is made to either match more bs and as, to pop excess as, or to read excess bs. More bs and as may be matched if a b is read and there is an a on the stack. In this case, the machine loops in B after reading the b and popping the a. There are more as than bs if a c is read and there is an a on the stack. In this case, the machine reads the c, pops the a, and transitions to a state, C, that reads cs as long as as can also be popped. There are more bs than as if a b is read and the bottom of the stack symbol

is the top stack element. In this case, the machine transitions to a state, D, to read the rest of the bs without changing the stack. The needed transitions are:

```
((B b (a)) (B ,EMP))
((B c (a)) (C ,EMP))
((B b (y)) (D (y)))
```

In C, the consumed input is a^i followed by b^j followed by c^k. In addition, 0<j<i, 0<k, and the stack contains i-j- k as and y. The state is documented as follows:

C: ci = (a^i b^j c^k) ∧ stack = (a^{i-j-k} y) ∧ i > j ∧ i,j,k > 0

As long as there are as on the stack and cs on the input tape, the machine reads a c and pops and a. If there are no more as on the stack the machine transitions to a state, E, to read the rest of the cs, if any, without modifying the stack. If there are no more cs to read the machine moves to a state, G, to pop the rest of the as. The needed transitions are:

```
((C c (a)) (C ,EMP))
((C c (y)) (E (y)))
((C z ,EMP) (G ,EMP))
```

In D, the consumed input is one or more as followed by 1 or more bs and there are more bs than as. In addition, the stack only contains the bottom of the stack symbol, because all as have already been matched. This state is documented as follows:

D: ci = (a^i b^j) ∧ stack = (y) and j > i ∧ i,j > 0

While there are bs to read, the machine loops in D. When a c is read the machine transitions to E to read the rest of the input. The needed transitions are:

```
((D b (y)) (D (y)))
((D c (y)) (E (y)))
```

In E, the consumed input consists of 1 or more as followed by 1 or more bs followed by 1 or more cs. In addition, the number of as is not equal to the number of bs and the stack only contains the bottom of the stack symbol. The state is documented as follows:

E: ci = (a^i b^j c^+) ∧ stack = (y), j != i ∧ i,j > 0

If there are cs to read the machine loops in E without modifying the stack. When the end of the input marker is read, the machine pops the bottom of the stack symbol and moves to, F, the final state to accept. The needed transitions are:

```
((E c (y)) (E (y)))
((E z (y)) (F ()))
```

Fig. 77 A deterministic pda for L = $\{a^m b^n c^p \mid m \neq n \wedge m,n,p>0\}$

```
#lang fsm

(define ambncp (make-ndpda '(S A B C D E F G)
                           '(a b c z)
                           '(a y)
                           'S
                           '(F)
                           `(((S a ,EMP) (A (a y)))
                             ((A a ,EMP) (A (a)))
                             ((A b (a)) (B ,EMP))
                             ((B b (a)) (B ,EMP))
                             ((B c (a)) (C ,EMP))
                             ((C c (a)) (C ,EMP))
                             ((C z ,EMP) (G ,EMP))
                             ((C c (y)) (E (y)))
                             ((B b (y)) (D (y)))
                             ((D b (y)) (D (y)))
                             ((D c (y)) (E (y)))
                             ((E c (y)) (E (y)))
                             ((E z (y)) (F ,EMP))
                             ((G ,EMP (a)) (G ,EMP))
                             ((G ,EMP (y)) (F ,EMP)))))

(check-equal? (sm-apply ambncp '(z)) 'reject)
(check-equal? (sm-apply ambncp '(a b b c c)) 'reject)
(check-equal? (sm-apply ambncp '(a a b b c c z)) 'reject)
(check-equal? (sm-apply ambncp '(a b a z)) 'reject)
(check-equal? (sm-apply ambncp '(a b b z)) 'reject)
(check-equal? (sm-apply ambncp '(a a a b b c c z)) 'accept)
(check-equal? (sm-apply ambncp '(a a a b c z)) 'accept)
(check-equal? (sm-apply ambncp '(a a a b b b b c c c z)) 'accept)
```

In G, the consumed input consists of 1 or more as followed by 1 or more bs followed by 1 or more cs followed by z. In addition, the number of as does not equal the number of bs and the stack contains 0 or more as and the bottom of the stack marker. The state is documented as follows:

G: ci = $(a^i\ b^j\ c^+\ z) \wedge$ stack = $(a^*\ y) \wedge$ i != j $\wedge$ i,j > 0

The machine loops in G popping as without reading any input. When the stack only contains the bottom of the stack marker, it is popped and the machine transitions to F to accept. The needed transitions are:

```
((G ,EMP (a)) (G ,EMP))
((G ,EMP (y)) (F ,EMP))
```

Fig. 78 Predicate invariants for ambncp I

```
;; word stack → Boolean
(define (S-INV ci s) (and (eq? ci '()) (eq? s '())))

(check-equal? (S-INV '(a a) '(a)) #f)
(check-equal? (S-INV '() '(a)) #f)
(check-equal? (S-INV '(a) '()) #f)
(check-equal? (S-INV '() '()) #t)

;; word stack → Boolean
(define (A-INV ci s)
  (and  (not (empty? ci)) (andmap (λ (r) (eq? r 'a)) ci)
        (eq? (last s) 'y)
        (= (length (takef s (λ (r) (eq? r 'a)))) (length ci))))

(check-equal? (A-INV '(a b) '(a)) #f)
(check-equal? (A-INV '(a) '(a a y)) #f)
(check-equal? (A-INV '(a a a) '(a a a y)) #t)
(check-equal? (A-INV '(a) '(a y)) #t)

;; word stack → Boolean
(define (B-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as)) (λ (r) (eq? r 'b))))
         (j (length ci-bs))
         (s-as (takef s (λ (r) (eq? r 'a))))]
    (and (equal? ci (append ci-as ci-bs)) (< 0 i) (< 0 j) (<= j i)
         (eq? (last s) 'y) (equal? s (append s-as '(y)))
         (= (length s-as) (- i j)))))

(check-equal? (B-INV '(b b) '(a)) #f)
(check-equal? (B-INV '(a a b b) '(a y)) #f)
(check-equal? (B-INV '(a a b b) '(y)) #t)
(check-equal? (B-INV '(a a a a b b) '(a a y)) #t)
```

In F, the consumed input equals the entire input on the tape consisting of 1 or more as followed by 1 or more bs followed by 1 or more cs followed by z. In addition, the stack is empty and the number of as and bs are not equal. The state is documented as follows:

F: ci = (a^i b^j c^+ z) ∧ stack = '() ∧ i != j,
∧ i,j > 0, final state

No transitions out of F are needed because the entire input has been read and the stack is empty.

The implementation of the machine is displayed in Fig. 77. Observe that the transition relation is deterministic. That is, for any given configuration there is at most one configuration that may be reached in one step. That is, there are no nondeterministic choices for the machine to make. You should also observe that the machine either accepts after reading all the input or rejects without reading all the input or not reaching the final state.

Fig. 79 Predicate invariants for ambncp II

```
;; word stack → Boolean
(define (C-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as)) (λ (r) (eq? r 'b))))
         (j (length ci-bs))
         (ci-cs (takef (drop ci (+ (length ci-as) (length ci-bs)))
                       (λ (r) (eq? r 'c))))
         (k (length ci-cs))
         (s-as (takef s (λ (r) (eq? r 'a))))]
    (and (equal? ci (append ci-as ci-bs ci-cs))
         (eq? (last s) 'y) (equal? s (append s-as '(y)))
         (= (length s-as)  (- i j k)) (> i j)  (> i 0)  (> j 0))))
(check-equal? (C-INV '(a a b c z) '(a y)) #f)
(check-equal? (C-INV '(a a b c) '(a y)) #f)
(check-equal? (C-INV '(a a b) '(a y)) #t)
(check-equal? (C-INV '(a a a a b c) '(a a y)) #t)

;; word stack → Boolean
(define (D-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as)) (λ (r) (eq? r 'b))))
         (j (length ci-bs))]
    (and (equal? ci (append ci-as ci-bs)) (equal? s '(y))
         (> j i) (> i 0) (> j 0))))
(check-equal? (D-INV '(a a b c c z) '(y)) #f)
(check-equal? (D-INV '(a) '(a y)) #f)
(check-equal? (D-INV '(a b b) '(y)) #t)
(check-equal? (D-INV '(a a b b b) '(y)) #t)

;; word stack → Boolean
(define (E-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as))
                       (λ (r) (eq? r 'b))))
         (j (length ci-bs))
         (ci-cs (takef (drop ci (+ (length ci-as) (length ci-bs)))
                       (λ (r) (eq? r 'c))))]
    (and (equal? ci (append ci-as ci-bs ci-cs))
         (>= (length ci-cs) 1) (equal? s '(y))
         (not (= j i)) (> i 0) (> j 0))))
(check-equal? (E-INV '(a b) '(a y)) #f)
(check-equal? (E-INV '(a a b) '(a y)) #f)
(check-equal? (E-INV '(a b b c) '(y)) #t)
(check-equal? (E-INV '(a a a b c c) '(y)) #t)
```

Fig. 80 Predicate invariants for ambncp III

```
;; word stack → Boolean
(define (G-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as)) (λ (r) (eq? r 'b))))
         (j (length ci-bs))
         (ci-cs (takef (drop ci (+ (length ci-as) (length ci-bs)))
                       (λ (r) (eq? r 'c))))
         (s-as (takef s (λ (r) (eq? r 'a))))]
    (and (equal? ci (append ci-as ci-bs ci-cs '(z)))
         (>= (length ci-cs) 1) (equal? s (append s-as '(y)))
         (not (= j i))
         (> i 0)
         (> j 0))))

(check-equal? (G-INV '(a) '(a y)) #f)
(check-equal? (G-INV '(a b c z) '(y)) #f)
(check-equal? (G-INV '(a a a b c z) '(a y)) #t)
(check-equal? (G-INV '(a a a b c c z) '(a a y)) #t)

;; word stack → Boolean
(define (F-INV ci s)
  (let* [(ci-as (takef ci (λ (r) (eq? r 'a))))
         (i (length ci-as))
         (ci-bs (takef (drop ci (length ci-as)) (λ (r) (eq? r 'b))))
         (j (length ci-bs))
         (ci-cs (takef (drop ci (+ (length ci-as) (length ci-bs)))
                       (λ (r) (eq? r 'c))))]
    (and (equal? ci (append ci-as ci-bs ci-cs '(z)))
         (>= (length ci-cs) 1)
         (empty? s)
         (not (= j i))
         (> i 0)
         (> j 0))))

(check-equal? (F-INV '(z) '()) #f)
(check-equal? (F-INV '(a a b b c cz) '()) #f)
(check-equal? (F-INV '(a a b z) '(y)) #f)
(check-equal? (F-INV '(a a b c z) '()) #t)
(check-equal? (F-INV '(a b b b c c z) '()) #t)
```

56.4 State Invariant Predicates

Writing invariant predicates flows from the conditions identified above. As general implementation strategy, whenever appropriate, local variables are defined for the expected components of the consumed input and the stack. In different predicates, local variables for the consumed input include those for the leading as, the bs following the as, the cs following the bs, the number of as (i.e., i), the number of bs (i.e., j) or the number of cs (i.e., k). The only

stack value defined with a local variable, whenever needed, is for the as on the stack. In each predicate, the subset of these local variables defined is used to implement a conjunction for the properties the consumed input and the stack are expected to have. The invariant predicates are displayed in Figs. 78 to 80.

56.5 Correctness

As always, we start by proving that the state invariant predicates always hold for computations that lead to accept. Subsequently, we prove that `L = L(ambncp)`. In the proof, `ci` is used to denote the consumed input and `P` is used to denote `ambncp`.

56.5.1 Proving State Invariants Hold

Theorem 2 *The state invariants hold when P is applied to w.*

The proof is by induction on, `n`, the number of transitions to consume the input word `w`.

Proof
When P starts, S-INV holds because ci = `'()` and the stack = `'()`. This establishes the base case.

Proof that invariants hold after each nonempty transition:

`((S a ,EMP) (A (a y)))`: By inductive hypothesis, `S-INV` holds. This means `ci = stack = '()`. Using this transition adds an `a` to ci and pushes `y` and then `a` onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+\ y)$. Thus, A-INV holds.

`((A a ,EMP) (A (a)))`: By inductive hypothesis, `A-INV` holds. This means ci = (a^+) and stack = $(a^+\ y)$. Using this transition adds an `a` to ci and pushes an `a` onto the stack. Therefore, after using this rule, ci = (a^+) and stack = $(a^+\ y)$. Thus, A-INV holds.

`((A b (a)) (B ,EMP))`: By inductive hypothesis, `A-INV` holds. This means ci = $(a^+) = (a^i)$ and stack = $(a^+\ y)$. Using this transition adds a `b` to ci and pops an `a` off the stack. Therefore, after using this rule, ci = $(a^i\ b^1) = (a^i\ b^j)$, stack = $(a^{i-1}\ y) = (a^{i-j}\ y)$ and $0 < j \leq i$. Thus, B-INV holds.

((B b (a)) (B ,EMP)): By inductive hypothesis, B-INV holds. This means ci = (a^i b^j), stack = (a^{i-j} y) and 0 < j ≤ i. Given that there is an a on the stack, j must be strictly less than i. That is, j < i. Using this transition adds a b to ci and pops an a off the stack. Therefore, after using this rule, ci = (a^i b^{j+1}), stack = ($a^{i-(j+1)}$ y), 0 < j+1, and j+1 ≤ i. Without loss of generality, we may think of j+1 as a new value for j. Thus, B-INV holds.

((B c (a)) (C ,EMP)): By inductive hypothesis, B-INV holds. This means ci = (a^i b^j), stack = (a^{i-j} y), and 0 < j ≤ i. Given that there is an a on the stack, j < i. Using this transition , adds a c to ci and pops an a. Therefore, after using this rule, ci = (a^i b^j c^1) = (a^i b^j c^k), stack = (a^{i-j-1} y) = (a^{i-j-k} y), i > j, and i,j,k > 0. Thus, C-INV holds.

((B b (y)) (D (y))): By inductive hypothesis, B-INV holds. This means ci = (a^i b^j), stack = (a^{i-j} y), and 0 < j ≤ i. Observe that if the stack does not contain as then j = i. Using this transition adds a b to ci without modifying the stack. Therefore, ci = (a^i b^{j+1}) and stack = (y) and j+1 > i and i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Therefore, D-INV holds.

((C c (a)) (C ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Using this transition adds a c to ci and pops an a from the stack. Therefore, ci = (a^i b^j c^{k+1}), stack = ($a^{i-j-(k+1)}$ y), i > j, and i,j,k+1 > 0. Without loss of generality, we may think of k+1 as a new value for k. Therefore, C-INV holds.

((C z ,EMP) (G ,EMP)): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Observe that i≠j and that c^k ∈c^+. Using this transition adds z to ci without changing the stack. Therefore, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.

((C c (y)) (E (y))): By inductive hypothesis, C-INV holds. This means ci = (a^i b^j c^k), stack = (a^{i-j-k} y), i > j, and i,j,k > 0. Note that c^k ∈c^+. Using this transition means that a c is added to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.

((D b (y)) (D (y))): By inductive hypothesis, D-INV holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a b to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^{j+1}), stack = (y), j+1 > i, i,j+1 > 0. Without loss of generality, we may think of j+1 as a new value for j. Thus, D-INV holds.

`((D c (y)) (E (y)))`: By inductive hypothesis, `D-INV` holds. This means ci = (a^i b^j), stack = (y), j > i, and i,j > 0. Using this transition adds a `c` to ci without changing the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, i,j > 0. Thus, E-INV holds.

`((E c (y)) (E (y)))`: By inductive hypothesis, `E-INV` holds. This means ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Using this transition adds a `c` to ci without modifying the stack. Therefore, after using this transition, ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Thus, E-INV holds.

`((E z (y)) (F ,EMP))`: By inductive hypothesis, `E-INV` holds. This means ci = (a^i b^j c^+), stack = (y), j ≠ i, and i,j > 0. Using this transition adds `z` to ci and pops `y` from the stack. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = `'()`, i ≠ j, and i,j > 0. Thus, F-INV holds.

`((G ,EMP (a)) (G ,EMP))`: By inductive hypothesis, `G-INV` holds. This means ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. The use of this transition informs us that stack = (a^+ y) (i.e., the stack is not empty and contains at least one `a`). This transition pops an `a` from the stack without reading any input. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. Thus, G-INV holds.

`((G ,EMP (y)) (F ,EMP))`: By inductive hypothesis, `G-INV` holds. This means ci = (a^i b^j c^+ z), stack = (a^* y), i ≠ j, and i,j > 0. The use of this transition informs us that stack = (y) (i.e., the stack does not contain any `a`s). This transition pops `y` off the stack without consuming any input. Therefore, after using this transition, ci = (a^i b^j c^+ z), stack = `'()`, i ≠ j, and i,j > 0. Thus, F-INV holds.

This establishes the inductive step and concludes the proof. □

56.5.2 Proving `L = L(P)`

As before, the proof is divided into two lemmas. The first is for when `w`∈`L` and the second for when `w`∉`L`

Lemma 3 *w*∈*L* ⇔ *w*∈*L(P)*

Proof
(⇒) Assume w∈L. This means that w = $a^m b^n c^p$, where m ≠ n and m,n,p > 0. Given that state invariants always hold, the computation on w looks as follows:

When m>n: (S (a^m b^n c^p z) EMP) ⊢ (A (a^{m-1} b^n c^p z) (a y))
⊢* (A (b^n c^p z) (a^m y))
⊢* (B (c^p z) (a^{m-n} y))
⊢* (F EMP EMP)

When m<n: (S (a^m b^n c^p z) EMP) ⊢ (A (a^{m-1} b^n c^p z) (a y))
⊢* (A (b^n c^p z) (a^m y))
⊢* (B (b^{n-m} c^p z) (y))
⊢* (D (c^p z) (y))
⊢* (F EMP EMP)

Therefore, w∈L(P).

(⇐) Assume w∈L(P). This means that M halts in F, the only final state, with an empty stack having consumed w. Given that the state invariants always hold we may conclude that w = $a^m b^n c^p$, where m ≠ n and m,n,p > 0. Therefore, w∈L. □

Lemma 4 *w∉L ⇔ w∉L(P)*

Proof
(⇒) Assume w∉L. This means that w ≠ $a^m b^n c^p$, m = n, or m,n,p ≯ 0. Given that state predicate invariants always hold, P cannot consume w and end in F with an empty stack. Therefore, w∉L(P).

(⇐) Assume w∉L(P). This means that P cannot transition into F with an empty stack having consumed w. Given that the state predicate invariants always hold, w≠$a^m b^n c^p$, m = n, or m,n,p ≯ 0. Therefore, w∉L. □

1 Design and implement a deterministic pda that decides L = {$a^i b^j$ | i ≠ j}.

2 Design and implement a deterministic pda that decides L = a^* ∪ {$a^i b^i$ | i > 0}.

3 Design and implement a deterministic pda that decides L = {$a^i b^i$| i ≥ 0} ∪ {$a^i c^i$| i ≥ 0}.

4 Design and implement a deterministic pda that decides L = {$a^i c b^j$ | i ≠ j} ∪ {$a^i d b^{2i}$ | i ≥ 0}.

5 Design and implement a deterministic pda that decides L = {$a^m b^n c^p$| n ≠ p}.

6 Design and implement a deterministic pda that decides $L = \{a^m b^n c^p | m = n \wedge m,n,p \geq 0\}$.

7 Design and implement a deterministic pda that decides $L = \{a^m b^n c^p | n = p \wedge m,n,p \geq 0\}$.

57 Are All Context-Free Languages Deterministic?

We return to our question regarding whether or not all context-free languages are deterministic. That is, does there exist, for every context-free language, a deterministic pda that decides it? Consider, for example, $L = \{w \mid w=b^* \vee w=b^n a^n$, where $n>0$. How can this language be decided by a deterministic pda? The machine must accumulate bs on the stack in case the input word is in $b^n a^n$ in order to match them with the as. If the input word is in b^* then the machine does not accumulate bs. The problem is that the machine cannot be ready to accept if the word is in b^* and at the same time be ready to check if the bs and the as match. Does this mean L is not a deterministic context-free language?

In order to design a deterministic pda for L, the machine needs the ability to detect the end of the input. In this manner, the machine may accumulate bs on the stack. If the end of the input is detected without reading an a then the machine moves to a state that pops all the bs and accepts. If an a is read then the machine moves to a state where it tries to match the as with the accumulated bs. It does not suffice to make the decision when the blank after the input word is read, because for a $w \in L$ a nondeterministic decision must be made to either accept/reject or continue the computation to pop the bs to empty the stack.

Instead, we shall change the meaning of accept for, M, a deterministic pda. A language,L, is deterministic context-free if Lz = L(M). The symbol z is used as an end-marker for the input word and must not be a symbol that may be used in any $w \in L$. When M is given an input word, z must be added to the end. It is straightforward to see that any deterministic context-free language is context-free. If M accepts Lz then a nondeterministic pda, M', may be constructed that nondeterministically "senses" the end of the input and transitions to states that consume no more input.

To answer the question of whether or not every context-free language is deterministic context-free, we shall build on two important concepts. First, we assume that if M accepts Lz then M is a simple pda (as defined in Sect. 48.1). That is, the transition rules pop at most 1 element and push at most 2 elements. In addition, when the machine starts a bottom-of-the-stack symbol

is pushed and just before ending the computation this symbol is popped. If the given `pda` is not simple, its transformation into a simple `pda` results in a machine that is still deterministic. Second, we define a configuration, `C = (Q w s)`, as a *dead end* if in zero or more steps it leads to a configuration, `E = (R w' s')`, in which no input has been consumed (i.e., `w = w'`) or the stack has not shrunk (i.e., |s'| ≥ |s|). If a `pda` does not have a dead-end configuration then it will eventually read all its input. This follows by a observing that the size of the stack may only be decreased a finite number of times before more input must be consumed. In the absence of a dead-end configuration, it is always the case that in the future of the computation input is consumed or the stack is shrunk.

If `M` is simple then determining if a configuration is a dead end only depends on the current state, the next element to read, and the top stack symbol. Let `Q` be a state, `a` be an input alphabet symbol, and `x` a stack alphabet symbol. `(Q a x)` is a dead-end if there does not exist a state `P` and stack symbol `y` such that `(Q a x)` $\vdash^*$ `(P EMP y)` nor `(Q a x)` $\vdash^*$ `(P a EMP)`. Let `D` be the set of dead-end configurations. Observe that `D` is finite.

Recall that context-free languages are not closed under complement (see Sect 53). To establish that not all context-free languages are deterministic, we shall prove that deterministic context-free languages are closed under complement.

Theorem 3 *Deterministic context-free languages are closed under complement.*

Proof

Assume L is a context-free language such that Lz is accepted by a deterministic simple `pda` M = (make-ndpda K Σ Γ S F R).

Intuitively, like done with `dfa`s (see Sect. 30.4), we would like to build a `pda` for L's complement by simply switching the role of M's accepting and rejecting states. Unfortunately, this does not work because `pda`s may reject without reading all the input. This may happen in two ways: M is in a configuration for which no transition rules apply or M is in a dead-end configuration.

We shall convert M into, M', an equivalent deterministic `pda` without dead-end configurations. M' is constructed as follows:

1. ∀ `(Q a x)`∈`D`, remove rules of the form `((Q a x) (H y))` in M and add `((Q a x) (U EMP))`, where U is a new non-accepting state.
2. ∀ a∈Σ, add the rule `((U a EMP) (U EMP))`
3. ∀ b∈Γ, add the rules `((U z EMP) (V EMP))` and `((V EMP b) (V EMP))`, where V is a new non-accepting state.

These transitions allow M' to read the remaining input without consulting the stack when in U (type 2 rules) and to empty the stack and reject upon reading z (type 3 rules).

M' is deterministic given that M is deterministic and the added rules do not make M' nondeterministic. Observe that M' rejects by moving to a non-accepting state whenever M does not read the entire input and that M' has no dead-end configurations. Thus, M' always reads the entire input.

Given that M' reads the entire input, reversing the role of accepting and non-accepting states results in a machine that accepts (Σ^*-L)z. That is, the resulting machine accepts L's complement. □

This a rather stunning result. It allows us to prove that there are context-free languages that are not deterministic. For example, consider $\mathtt{L} = \{\mathtt{a}^\mathtt{i}\mathtt{b}^\mathtt{j}\mathtt{c}^\mathtt{k} \mid \mathtt{i} \neq \mathtt{j} \vee \mathtt{i} \neq \mathtt{k}\}$. Assume L is a deterministic context-free language. The theorem above informs us that L's complement, $\bar{\mathtt{L}}$, is also a deterministic context-free language. Recall that the intersection of a context-free language and a regular language is a context-free language (proven in Sect. 54). Observe that $\bar{\mathtt{L}} \cap \mathtt{a}^*\mathtt{b}^*\mathtt{c}^* = \mathtt{a}^\mathtt{n}\mathtt{b}^\mathtt{n}\mathtt{c}^\mathtt{n}$. This is a language that we know is not context-free. Thus, our assumption is wrong and L is not a deterministic context-free language.

Clearly, the deterministic context-free languages are a proper subset of the context-free languages. This means that, in the context of pdas, nondeterminism is more powerful than determinism. This is a sharp contrast to what we discovered for finite-state machines: nondeterminism is not more powerful. There are profound consequences for modern software development. For instance, interpreters and compilers must parse a given program. That is, given a program they try to build a parse tree based on a programming language's grammar. If this grammar is deterministic context-free then it is relatively straightforward to write a parser. Such a grammar, as we have seen, may not be deterministic context-free. For such grammars, it becomes necessary to convert them to a deterministic pushdown automata that can decide if the given program is or not in the set of valid programs for the programming language. If such a transformation is possible then it usually involves using heuristic rules that may (or may not) be useful to achieve it. The techniques used are beyond the scope of this book. Nonetheless, you ought to be aware that nondeterministic pdas are, computationally speaking, more powerful than deterministic pdas and that this makes software development more difficult.

58 Closure Properties of Deterministic Context-Free Languages

58.1 Union

Section 49 established that context-free languages are closed under union. Surprisingly, the same is not true for deterministic context-free languages. Deterministic context-free languages are not closed under union. This is established in the following theorem.

Theorem 4 *Deterministic context-free languages are not closed under union.*

Proof

Assume deterministic context-free languages are closed under union.

Consider the following deterministic context-free languages:

- $L = \{a^m b^n c^p | \ m \neq n \wedge m,n,p > 0\}$
- $S = \{a^m b^n c^p | \ n \neq p \wedge m,n,p > 0\}$

In Sect. 56, we established that L is a deterministic context-free language. Establishing that S is a deterministic context-free language was left as an exercise.

Let LUS = L ∪ S. Given our assumption, LUS is a deterministic context-free language.

In Sect. 57, we established that deterministic context-free languages are closed under complement. Thus, the complement of LSU, NOT-LSU, is also a deterministic context-free language. NOT-LSU is the union of the following two languages:

- $\{a^m b^n c^p | \ m = n = p \wedge m,n,p > 0\}$
- {w | w∈{a b c}* ∧ w has letters out of order ∨ w is missing at least one element in {a b c}}

In Sect. 54, we proved that the intersection of a context-free language and a regular language is a context-free language. This means that NOT-LSU ∩ $a^* b^* c^*$ is a context-free language. Observe, however, that this language is $a^n b^n c^n$, where n>0. This is not a context-free language.[12] Thus, our assumption is wrong and deterministic context-free languages are not closed under union. □

[12] To prove this use the Pumping Theorem for context-free languages as done for $a^n b^n c^n$, where n≥0, in Sect. 52.3.

58.2 Intersection

Like their context-free counterparts, deterministic context-free languages are not closed under intersection. This is established in the following theorem.

Theorem 5 *Deterministic context-free languages are not closed under intersection.*

Proof

Assume deterministic context-free languages are closed under intersection.

Consider the following languages:

- $L = \{a^m b^n c^p |\ m = n \wedge m,n,p \geq 0\}$
- $S = \{a^m b^n c^p |\ n = p \wedge m,n,p \geq 0\}$

Both of these languages are deterministic context-free (see exercises after Sect. 56).

The intersection of L and S is $\{a^n b^n c^n |\ n \geq 0\}$. In Sect. 52.3, we proved that this language is not context-free. Given that it is not context-free, it cannot be deterministic context-free. Therefore, our assumption is wrong and deterministic context-free languages are not closed under intersection. □

8 Prove that the intersection of a regular language and a deterministic context-free language is a deterministic context-free language.

9 Prove that deterministic context-free languages are not closed under concatenation.

10 Prove that deterministic context-free languages are not closed under Kleene Star.

Part IV
Context-Sensitive Languages

Chapter 15
Turing Machines

We have seen that dfas, ndfas, and pdas are incapable of recognizing rather simple languages like L = $\{a^n b^n c^n \mid n \geq 0\}$. Any reader of this textbook, however, can write a program in their favorite programming language to decide if a given word is in L. For this reason, these models of computation cannot be considered a general model for modern computers. We need a more general model that is capable of deciding L and other more complex languages.

The new computation model that we shall study will not be replaced by a more powerful model like pdas replaced dfas. To date, no one has been able to strengthen the new computation model that we shall study. All attempts to strengthen the new model have failed to offer any new computation power. The new model is called a *Turing machine* named after its inventor Alan Turing. A Turing machine (tm), like previous automatons, has a control mechanism to track the current machine state, an input tape, and a reading head. Unlike previous automatons, the head may move right or left on the input tape and may write to the tape. You may think of writing on the tape as mutation (i.e., assignment) that changes the value on the tape. The value overwritten is lost forever, and after the mutation, only the new value is accessible.

We shall do much more than simply decide or accept a language with a Turing machine. We shall also compute the value of functions. Consider, for example, the following function:

```
;; number number → number
;; Purpose: Add the given numbers
(define (plus a b) (+ a b))
```

We shall see that a Turing machine, just like this function, can add two numbers. In fact, it is theorized that a Turing machine can compute anything

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_15

that is computable. In this sense, Turing machines are more powerful than any computer in existence today.

59 Turing Machine Definition

We shall start by thinking of Turing machines as language recognizers. Like other automatons, a Turing machine is a type in `FSM`. A Turing machine language recognizer is an instance of:

```
(make-tm K Σ R S F Y)
```

The components of a Turing machine language recognizer are described as follows:

K The states.
Σ The input alphabet.
R The transition *function*.
S The starting state.
F The final states.
Y The accepting final state

`K`, Σ, `S`, and `F` are defined as in previous automatons. In addition to Σ's elements, there are two special symbols that may appear of the input tape: `LM` and `BLANK` (both `FSM` constants). `LM` may only appear in the input tape's leftmost position. It is the input tape's left-end marker. `BLANK` may appear anywhere on the input tape (except the leftmost position) and represents a blank (a position in the input tape with nothing written in it). This definition specifies `R` as a transition function. That is, the `tm` is deterministic. Later, we shall relax this condition and allow `R` to be a transition relation. For now, however, we shall design deterministic Turing machines.

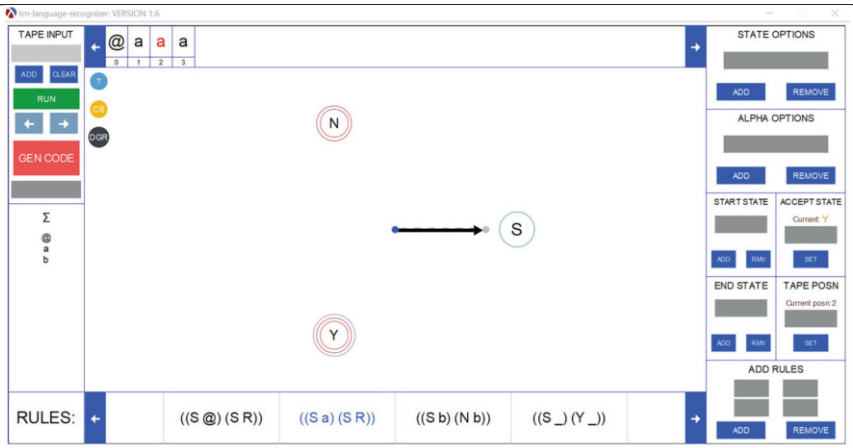
A deterministic Turing machine language recognizer requires two final states usually named `Y` and `N`. If the machine reaches `Y` it halts and accepts the input even if it has not consumed all of it. If the machines reaches `N`, it halts and rejects the input even if, once again, it has not consumed all of it. This means that when a Turing machine reaches a final state, it halts and performs no more transitions.

A Turing machine rule, `tm`-rule, is an element of:

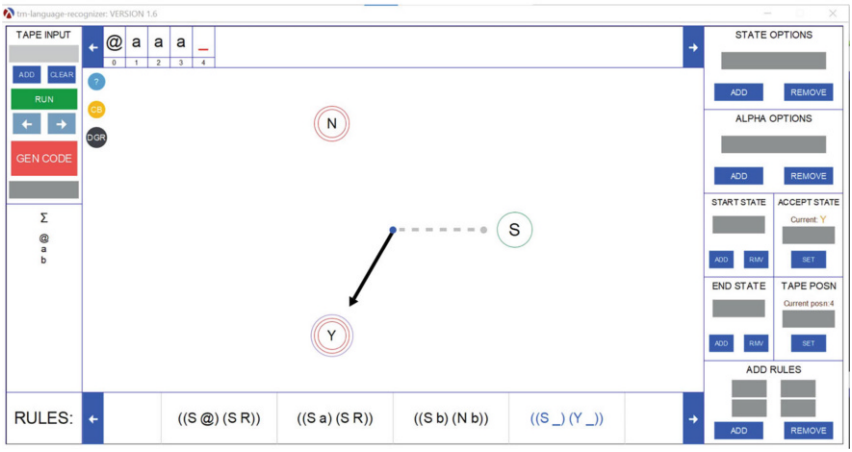
```
(list (list N a) (list M A))
```

`N` is non-halting state (i.e., `N`$\in$`{K-F}`), `a`$\in\{\Sigma \cup$ `{LM BLANK}}`, `M`$\in$`K` (which may or may not equal `N`), and `A` is an `action`$\in\{\Sigma \cup$ `{RIGHT LEFT}`$\}$. If `A`$\in \Sigma$ then the machine writes `A` in the tape position under the tape's head. This is equivalent to mutation. That is, the tape is permanently changed, and the value overwritten is no longer accessible. If `A`$\in$`{RIGHT LEFT}`$\}$, then the machine moves the tape's head. `RIGHT` is an `FSM` constant to move the head right, and `LEFT` is an `FSM` constant to move the head left. The only exception

Fig. 81 Visualization of `tm` configurations



(a) A `tm` non-halting configuration.



(b) A `tm` halting configuration.

is when `LM` is read. When `LM` is read, the `tm` must move the tape's head right (regardless of the state it is in) and may not overwrite `LM`. Observe that with such a definition, the `tm` cannot "fall off" the left end of the input tape. The constructor `make-tm` automatically adds `LM` to Σ and automatically adds the rules to move the tape's head right when `LM` is read. These rules, of course, are only added for non-final states. It is pointless to add such rules for final states because a Turing machine always halts upon reaching a final state.

A Turing machine configuration is a triple: `(s n t)`. The machine's current state is denoted by `s`. The position of the head on the tape is denoted by `n`. The input tape is denoted by `t`. The input tape has an infinite number of blanks to the right of the last non-blank in the tape. When a configuration

is displayed, only the "touched" part of the tape is displayed. The touched part of the input tape includes the left-end marker and anything specified in the initial tape value including blanks. Let us call the largest tape position specified in the initial tape value `i`. If the tape's head has never gone to the right of `i`, then the part of the tape displayed is `t[0..i]`. If the tape's head has, at any time during the computation, moved to the right of `i`, then let us call the rightmost position reached `j`. The part of the tape displayed is `t[0..j]`. Figure 81a displays the `FSM` control view visualization for `(S 2 (@ a a a))`. `LM` is in position 0, and the head is on position 2 of the tape (i.e., the second `a`). Figure 81b displays the `FSM` control view visualization for `(Y 4 (@ a a a _)`. `LM` is in position 0, and the head is on position 4 of the tape (i.e., the first blank after the last `a`). Observe that the final states are in double red circles and that the accepting final state is also enclosed in a blue circle.

A computation for a `tm`, `M`, is denoted by a list of configurations that `M` traverses. A transition made by the machine is denoted, as done for other automatons, using $\vdash$. $\texttt{C}_\texttt{i} \vdash \texttt{C}_\texttt{j}$ is valid for `M` if and only if `M` can move from $\texttt{C}_i$ to $\texttt{C}_j$ using a single transition. Zero or more moves by `M` are denoted, as before, using $\vdash^*$. $\texttt{C}_\texttt{i} \vdash^* \texttt{C}_j$ is valid for `M` if and only if `M` can move from $\texttt{C}_i$ to $\texttt{C}_j$ using zero or more transitions. A word, `w`, is accepted by a `tm` language recognizer if it reaches the final accepting state (i.e., `Y` in the definition above). Otherwise, `w` is rejected.

The observers `sm-apply` and `sm-showtransitions` may consume an optional third argument that is a natural number. This number denotes the starting position of the head on the input tape when the machine starts. If this optional argument is not provided, then the default initial head position, 0, is used. It is opportune to note that `sm-showtransitions` only returns `'reject` when the given word is rejected.

A Turing machine language recognizer's execution may be observed using `sm-visualize`. As you may already suspect, the execution may only be visualized when the given word is in the machine's language. State invariant predicates take as input the "touched" part of the input tape and the position, `i`, of the input tape's next element to read. The predicate asserts a condition about the touched input that must hold which may or may not be in relation to the head's position. For language recognizers, it is important to remember that when the machine's head is at position `i`, the tape element at `i` has not been read. Finally, to add a blank to the input, you type `BLANK` in the tape input box in the left column.

60 A Turing Machine for `L = a*`

To familiarize ourselves with designing Turing machines, we start by designing a language recognizer for `L = a*`. We shall follow the steps of the design recipe for state machines in Fig. 20.

60.1 Name, Alphabet, and Tests

A descriptive name for the tm language recognizer is a*, and the input alphabet is {a b}. Unlike earlier automatons, a tm's head may start on any tape position. Therefore, it is important to specify a precondition. A tm precondition is an assertion about the machine's initial configuration. That is, it is an assertion about the contents of the tape and the position of the head when the machine starts its execution. For instance, the precondition for a* is stated as follows:

```
;; PRE: tape = LMw ∧ i = 0
```

The precondition states that the tape contains the left-end marker followed by, w, the input word. It is assumed that everything to the right of w are blanks. The precondition also states that a*'s head starts on position 0 (i.e., over LM). Think of the precondition as establishing a contract with the machine's user stating that if the machine starts in the correct configuration, then it works correctly. No claim is made if the machine is not started in the correct configuration.

The machine ought to decide L = a*. This means that the machine needs to be tested with both words in and not in L. We test a* as follows:

```
;; Tests for a*
(check-equal? (sm-apply a* `(,LM a a a b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM)) 'accept)
(check-equal? (sm-apply a* `(,LM a a a)) 'accept)
```

60.2 Conditions and States

We denote the contents of the tape from position i to position j by tape[i..j]. The tm may remain in its starting state as long as it has only read as. If a blank is read, then it may move to the final accepting state because the input word is in L. If a b is read, then the machine may move to the final rejecting state because the input word is not in L.

We may document the states as follows:

```
;; States (i = head's position)
;;   S: tape[1..i-1] only contains as, starting state
;;   Y: tape[i] = BLANK and tape[1..i-1] only contains as,
;;      final state
;;   N: tape[i] is b, final state
```

In this example, the role of each state is defined in terms of the input's tape contents and the head's position.

Fig. 82 A `tm` to decide `L = a*`

```
;; States (i = head's position)
;;   S: tape[1..i-1] only contains as, starting state
;;   Y: tape[i] = BLANK and tape[1..i-1] only contains as, final state
;;   N: tape[1..i-1] contains a b, final state

;; L = a*
;; PRE: tape = LMw_ AND i = 0
(define a* (make-tm '(S Y N)
                    '(a b)
                    `(((S a) (S ,RIGHT))
                      ((S b) (N b))
                      ((S ,BLANK) (Y ,BLANK)))
                    'S
                    '(Y N)
                    'Y))

;; Tests for a*
(check-equal? (sm-apply a* `(,LM a a a b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM)) 'accept)
(check-equal? (sm-apply a* `(,LM a a a)) 'accept)
```

60.3 Transition Function, Implementation, and Testing

The machine starts in `S` with the head at position 0. This means that the next element to read is `LM`, and the machine includes a rule (automatically added by the constructor) to remain in `S` and move to the right:

```
`((S ,LM) (S R))
```

After reading `LM`, the machine is in `S` and may read an `a`, a `b`, or `BLANK`. If an `a` is read, then only `a`s have been read. The machine may remain in `S` to continue reading `w` by moving the head right because `S`'s condition holds. Thus, the needed rule is:

```
((S a) (S R))
```

If a `b` is read, then `w` does not contain only `a`s. This means that $w \notin L$ and the machine may reject by moving to `N` and writing `b`. Thus, the needed rule is:

```
((S b) (N b))
```

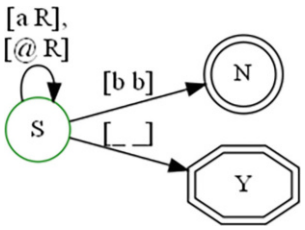
Finally, if `BLANK` is read, then `w` has been entirely read, and it only contains `a`s. The machine may accept by moving to `Y` and writing `BLANK` on the tape. Thus, the needed transition rule is:

```
`((S ,BLANK) (Y ,BLANK))
```

The `a*` implementation is displayed in Fig. 82. It is opportune to examine the machine's alphabet and transition function:

```
> (sm-sigma a*)
'(@ a b)
> (sm-rules a*)
'(((S @) (S R))
  ((S a) (S R))
  ((S b) (N b))
  ((S _) (Y _)))
```

As expected, the constructor adds `LM` to the input alphabet and adds the rules (in this case just 1) to move to the right when `LM` is read. Inspecting the transition diagram using `sm-graph` yields:



The labels on the edges contain the read element and the action taken. For instance, `[a R]` means that `a` is read and the head moves to the right, and `[b b]` means that `b` is read and `b` is written to the tape at the current head position (i.e., in this case, the `b` read is overwritten with a `b`).

Running the tests reveals that they all pass. In addition to running the tests, we can perform random testing:

```
> (sm-test a* 10)
'(((@ _) accept)
  ((@ a a b a b a b a) reject)
  ((@ b b a a) reject)
  ((@ a a) accept)
  ((@ b b b) reject)
  ((@ b b b b) reject)
  ((@ a) accept)
  ((@ a a b b a) reject)
  ((@ b a a b b a b b b) reject)
  ((@ b a a a a b b a b) reject))
```

A visual inspection of the results reveals that the random tests produce the correct result. This gives us cautious optimism that the machine is correctly implemented.

60.4 Invariant Predicates

The invariant for `S` verifies that all tape elements from position 1 to the given position minus 1 are `a`s. When the machine starts, of course, nothing is read, and the invariant for `S` must hold. This means it must return true when the given tape position is `i = 0`. If the given position `i` $\neq$ `0`, then `andmap` may be used to verify that the first `i - 1` elements in the rest of, `t`, the tape are `a`s (observe that neither `LM` at position 0 nor the `ith` are tested). Finally, tests are written using `t` and `i` values that illustrate when false and true ought to be returned. The invariant is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a
(define (S-INV t i)
  (or (= i 0)
      (andmap (λ (s) (eq? s 'a))
              (take (rest t) (sub1 i)))))

;; Tests for S-INV
(check-equal? (S-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (S-INV `(,LM a a b a a) 4) #f)
(check-equal? (S-INV `(,LM) 0) #t)
(check-equal? (S-INV `(,LM b) 1) #t)
(check-equal? (S-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (S-INV `(,LM a a a a ,BLANK) 5) #t)
```

The invariant for `Y` verifies that the tape value at the given position is `BLANK` and that everything from position 1 to the given position minus 1 on the tape is an `a`. The implementation is fairly straightforward after designing and implementing `S-INV`. We must make sure, however, that the tests use tape and position values that make the invariant hold and not hold. The implementation is:

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a ∧
;;          tape[i] = BLANK
(define (Y-INV t i)
  (and (eq? (list-ref t i) BLANK)
       (andmap (λ (s) (eq? s 'a))
               (take (cdr t) (sub1 i)))))

;; Tests for Y-INV
(check-equal? (Y-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (Y-INV `(,LM a b a ,BLANK) 2) #f)
(check-equal? (Y-INV `(,LM a b a ,BLANK) 4) #f)
(check-equal? (Y-INV `(,LM ,BLANK) 1) #t)
(check-equal? (Y-INV `(,LM a a a ,BLANK) 4) #t)
```

The invariant for N verifies that the given tape at the given position is a b. The implementation and tests are:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[i] = b
(define (N-INV t i) (eq? (list-ref t i) 'b))

;; Tests for N-INV
(check-equal? (N-INV `(,LM ,BLANK) 1) #f)
(check-equal? (N-INV `(,LM a a ,BLANK) 3) #f)
(check-equal? (N-INV `(,LM a b a b ,BLANK) 4) #t)
(check-equal? (N-INV `(,LM b b b) 2) #t)
(check-equal? (N-INV `(,LM a a a a b ,BLANK) 5) #t)
```

60.5 Correctness

We start by proving that the state invariants hold when a* is applied to w as stated in the following theorem:

Theorem 1 *State invariants hold when a* is applied to w.*

The proof, as before, is done by induction on, n, the number of steps taken by a*.

Proof

Base case: n = 0
If no steps are taken, a* must be in S, and by precondition, the head's tape position must be 0. Thus, S-INV holds.

Inductive Step:
Assume: State invariants hold for a computation of length n = k Show: State invariants hold for a computation of length n = k + 1

Observe that n = k + 1 means that w ≠ EMP. Let w = xcy, such that x,y∈ Σ^*, |x|=k, and c∈{Σ ∪ {BLANK}}. The first k + 1 steps may be described as follows:

(S 0 xcy) $\vdash^*$ (S i cy) $\vdash$ (B j y), where B∈{S Y N}

That is, the first k elements of w are traversed in S. Traversing the k + 1 element is done in one step that may take the machine to any state. We must show that the state invariants hold for k + 1 transition. We make an argument for each rule that may be used:.

((S @) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head from position 0 to position 1. Observe that [1..0] is empty and, therefore, everything in that tape interval is an `a`. Thus, S-INV holds after using this rule.

((S a) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head one position to the right from i to i+1. The inductive hypothesis informs us that all tape elements in [1..i-1] are `a`. The use of this rule means that the tape's ith element is `a`. This means that all tape elements in [1..i] are `a`. Thus, S-INV holds after using this rule.

((S b) (N b)): By inductive hypothesis, S-INV holds. Using this rule moves the machine to N without moving the head from position i and leaving the tape unchanged. This means that the tape's ith element is `b`. Thus, N-INV holds.

((S BLANK) (Y BLANK)): By inductive hypothesis, S-INV holds. Using this rule moves the machine to Y without moving the head from position i and leaving the tape unchanged. This means that after using this rule, the tape's elements in positions [1..i-1] are `a` and that the ith element is `BLANK`. Thus, Y-INV holds. □

Armed with the knowledge that the state invariants always hold, we may proceed to prove the following theorem:

Theorem 2 *$L = L(a^*)$*

As before, the proof is divided into two lemmas.

Lemma 1 *$w \in L \Leftrightarrow w \in L(a^*)$*

Proof
($\Rightarrow$) Assume w $\in$ L. This means w consists of 0 or more `a`s. Given that the state invariants always hold, the only state that `a*` may be in after consuming w is `Y`. Thus, w $\in$ L(a*).

($\Leftarrow$) Assume w $\in$ L(a*). Given that state invariants always hold, this means that w contains only `a`s. Therefore, w $\in$ L. □

Lemma 2 *$w \notin L \Leftrightarrow w \notin L(a^*)$*

Proof
($\Rightarrow$) Assume w $\notin$ L. This means w contains a `b`. Given that the state invariants always hold, `a*` cannot be in `Y` after consuming w. Thus, w $\notin$ L(a*).

($\Leftarrow$) Assume w $\notin$ L(a*). Given that state invariants always hold, this means that w contains a `b`. Therefore, w $\notin$ L. □

This concludes the design, implementation, and correctness argument for a Turing machine to decide a*. It is common for beginners with Turing machines to be puzzled by the following behavior:

```
> (sm-apply a* `(a a ,BLANK a a))
'accept
```

How can we claim that the machine is correct when it accepts a word that does not only contain as? This highlights the importance of preconditions. Recall the precondition for a*:

```
;; PRE: tape = LMw_ AND i = 0
```

The precondition clearly states that when the machine starts, the tape must contain the left-end marker followed by w. The word, w, must be a member of Σ^*. This means that every element of w is in Σ. Therefore, w may not contain BLANK. Thus, `(a a ,BLANK a a) is not valid input for a*, and as stated on the onset of our design, we make no claims about how the machine behaves when given an invalid tape.

1 Let Σ = {a b}. Design and implement a tm language-recognizer for L = {w | w has an even number of as and an even number of bs}.

2 Let Σ = {a b}. Design and implement a tm language-recognizer for L = {w | the number of as in w is divisible by 3}.

3 Your hacker friend claims that the following may be the precondition for a*:

```
;; PRE: tape = LMw AND i = 1
```

Is she correct? Justify your answer.

61 Nondeterministic Turing Machines

A Turing machine language recognizer may be nondeterministic. The only difference with a deterministic Turing machine language recognizer is that its rules define a transition *relation* (not a function). To illustrate the design and

implementation of a nondeterministic Turing machine language recognizer, we consider the language L = a* ∪ a*b.

61.1 Name, Alphabet, and Tests

The machine is named a*Ua*b, and the input alphabet is Σ = {a b}. The precondition is:

```
;; PRE: tape = LMw AND i = 1
```

That is, we shall design based on the tape containing the left-end marker followed by, w, the input word and on the tape's head being over w's first element if it exists. If w is empty, then the tape's head is on the first blank after the left-end marker. Either way, note that whatever is under the tape's head has not been read.

The test are written using words that are in and that are not in a* or a*b. For each test, the given head position is 1 to satisfy the precondition. Sample tests are:

```
;; Tests for a*Ua*b
(check-equal? (sm-apply a*Ua*b `(,LM b b)     1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM a a b a) 1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM ,BLANK)  1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a b)     1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a)   1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a b) 1) 'accept)
```

61.2 Conditions and States

When the machine starts in S, nothing has been read. If the input word is empty, the machine moves to accept. If the first element is an a, then the machine nondeterministically moves to a state A to read a* or to a state B to read a*b. Upon reading a* in A, the machine may accept. Upon reading a*b in B, the machine moves to state C to determine if the end of the input word has been reached and then moves to either accept or reject.

To accept, the machine must reach the first blank after the input word. If in A, the machine may move to accept upon reading a blank. If in B, on the other hand, the machine cannot move to accept upon reading a b because the end of the input word may not have been reached. The state C reflects that a*b has been read. From C, the machine may accept upon reading a blank and reject otherwise.

The states may documented as follows:

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] in a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* nor a*b, final state
```

61.3 Transition Function, Implementation, and Testing

When in S, upon reading a blank, the machine moves to Y, without moving the head nor changing the tape, because the empty word is in a^*. Upon reading an a, the machine nondeterministically decides to move to A to read a word in a^* or to B to read a word in a^*b. Upon reading a b, the machine moves to C, given that tape[1..i-2] has zero as followed by a b. In the three latter cases, the machine moves the head right to read, if any, the next element of the input. The needed transitions are:

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

When in A upon reading a blank, the machine may move to Y and accept because the input word is an a^* element. Upon reading an a, the machine remains in A and moves the head to the right. The needed transitions are:

```
((A a) (A ,RIGHT))
((A ,BLANK) (Y ,BLANK))
```

When in B upon reading an a, the machine remains in B to read more as. Upon reading a b, the machine moves to C to determine if the end of the input word has been reached. In both cases, the machine moves the head right to read, if any, the next element of the input. The needed transitions are:

```
((B a) (B ,RIGHT))
((B b) (C ,RIGHT))
```

When in C upon reading a blank, the machine may move to Y and accept because the input word is in a^*b. Upon reading an a or a b, the machine may

Fig. 83 The nondeterministic `tm` language recognizer for `L = a* U a*b`

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] = a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* or a*b, final state

;; L = a* U a*b
;; PRE: tape = LMw AND i = 1
(define a*Ua*b (make-tm '(S A B C Y N)
                        `(a b)
                        `(((S ,BLANK) (Y ,BLANK))
                          ((S a) (A ,RIGHT))
                          ((S a) (B ,RIGHT))
                          ((S b) (C ,RIGHT))
                          ((A a) (A ,RIGHT))
                          ((A ,BLANK) (Y ,BLANK))
                          ((B a) (B ,RIGHT))
                          ((B b) (C ,RIGHT))
                          ((C a) (N a))
                          ((C b) (N b))
                          ((C ,BLANK) (Y ,BLANK)))
                        'S
                        '(Y N)
                        'Y))

;; Tests for a*Ua*b
(check-equal? (sm-apply a*Ua*b `(,LM b b) 1)     'reject)
(check-equal? (sm-apply a*Ua*b `(,LM a a b a) 1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM b) 1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM ,BLANK) 1)  'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a b) 1)     'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a) 1)   'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a b) 1) 'accept)
```

move to `N`, move the head to the right, and reject because the input word is not in `a*b`. The needed transitions are:

```
((C a) (N ,RIGHT))
((C b) (N ,RIGHT))
((C ,BLANK) (Y ,BLANK)))
```

The implementation of `a*Ua*b` is displayed in Fig. 83. Running the tests reveals that they all pass. The machine is also validated using random testing:

```
> (sm-test a*Ua*b 10)
'(((@ _) accept)
  ((@ a a b) accept)
  ((@ b) accept)
  ((@ a b b b a a b a) reject)
  ((@ b b a) reject)
  ((@ b a a a b a) reject)
  ((@ a b a b b) reject)
  ((@ a b b b) reject)
  ((@ b b b) reject)
  ((@ a a) accept))
```

A visual inspection of the results reveals that all the randomly generated words are correctly decided.

61.4 Invariant Predicates

The invariant predicate for `S` must determine that nothing on the tape has been read. In accordance with the precondition for `a*Ua*b`, this means that the given head position must be 1. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that no tape elements read
(define (S-INV t i) (= i 1))

(check-equal? (S-INV `(,LM a a) 2) #f)
(check-equal? (S-INV `(,LM a a b) 1) #t)
```

The invariant predicate for `A` must determine that only `a`s have been read and that the head's position is at least 1. This means that the tape positions in `[1..i-1]` only contain `a`s, where `i` is the head's position. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i-1] only has a
(define (A-INV t i)
  (and (>= i 2)
       (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))

(check-equal? (A-INV `(,LM ,BLANK ,BLANK) 2) #f)
(check-equal? (A-INV `(,LM a) 0) #f)
(check-equal? (A-INV `(,LM a b a a a) 3) #f)
(check-equal? (A-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (A-INV `(,LM a a a b) 4) #t)
```

The invariant predicate for `B` is the same as the invariant predicate for `A`. The predicate may be implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i-1] only has a
(define (B-INV t i)
  (and (>= i 2)
       (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))

(check-equal? (B-INV `(,LM ,BLANK ,BLANK) 2) #f)
(check-equal? (B-INV `(,LM a) 0) #f)
(check-equal? (B-INV `(,LM a b a a a) 5) #f)
(check-equal? (B-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (B-INV `(,LM a a a b) 3) #t)
```

The invariant predicate for `C` must determine that only `a`s followed by a `b` have been read. This means that the tape positions `[1..i-2]` only contain `a`s and position `i-1` contains a `b`, where `i` is the head's position. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i-2] has only a and
;;          tape[i-1] = b
(define (C-INV t i)
  (and (>= i 2)
       (andmap (λ (s) (eq? s 'a)) (take (rest t) (- i 2)))
       (eq? (list-ref t (sub1 i)) 'b)))

(check-equal? (C-INV `(,LM ,BLANK ,BLANK) 2) #f)
(check-equal? (C-INV `(,LM a a b b a) 5) #f)
(check-equal? (C-INV `(,LM a a b b a) 4) #t)
(check-equal? (C-INV `(,LM b ,BLANK) 2) #t)
```

The invariant predicate for `Y` must determine that tape positions `[1..i-1]` only contains a blank, only contains `a`s, or contains `a`s followed by a `b`, where `i` is the head's position. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[i] = BLANK and
;;          tape[1..i-1] = a* or tape[1..i-1] = a*b
(define (Y-INV t i)
  (or (and (= i 2) (eq? (list-ref t (sub1 i)) BLANK))
      (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))
      (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
             (back (takef (drop t (add1 (length front)))
                          (λ (s) (not (eq? s BLANK)))))]
        (equal? back '(b)))))
```

```
(check-equal? (Y-INV `(,LM a a a) 1) #f)
(check-equal? (Y-INV `(,LM b a a) 3) #f)
(check-equal? (Y-INV `(,LM a a a a ,BLANK) 5) #t)
(check-equal? (Y-INV `(,LM a a a a a b,BLANK) 7) #t)
```

Finally, the invariant predicate for `N` must determine that tape positions `[1..i-1]` do not contain only `a`s nor contain `a`s followed by a `b`, where `i` is the head's position. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[1..i-1] != a* or a*b
(define (N-INV t i)
  (and (not (andmap (λ (s) (eq? s 'a))
                    (take (rest t) (sub1 i))))
       (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
              (back (takef (drop t (add1 (length front)))
                           (λ (s) (not (eq? s BLANK)))))]
         (not (equal? back '(b))))))

(check-equal? (N-INV `(,LM ,BLANK) 1) #f)
(check-equal? (N-INV `(,LM b ,BLANK) 1) #f)
(check-equal? (N-INV `(,LM a b ,BLANK) 3) #f)
(check-equal? (N-INV `(,LM a a a ,BLANK) 4) #f)
(check-equal? (N-INV `(,LM a b a ,BLANK) 4) #t)
(check-equal? (N-INV `(,LM a a a b b ,BLANK) 5) #t)
```

61.5 Correctness

We start by proving that the state invariants hold when `a*Ua*b` is applied to `w` as stated in the following theorem:

Theorem 3 *State invariants hold when a*Ua*b is applied to w.*

The proof, as before, is done by induction on, `n`, the number of steps taken by `a*Ua*b`. Let `a*Ua*b` = `(make-tm K Σ R S F Y)`.

Proof

Base case: n = 0
If no steps are taken, a*Ua*b may only be in S. By precondition, the head's position is 1. This means S-INV holds.

Inductive Step:
Assume: State invariants hold for a computation of length n = k
Show: State invariants hold for a computation of length n = k + 1

Let w = xcy, such that x,y$\in \Sigma^*$, |x|=k, and c$\in\{\Sigma \cup \{\text{BLANK}\}\}$. The first k + 1 steps may be described as follows:

$$\texttt{(S 1 xcy)} \vdash^* \texttt{(U r xcy)} \vdash \texttt{(V s xcy), where V}\in\texttt{K} \wedge \texttt{U}\in\texttt{K-\{N Y\}}$$

That is, the first k transitions take the machine to state U and move the head to position r without changing the contents of the tape. The k + 1 transition takes the machine to state V and leaves the head in position s without changing the contents of the tape. We must show that the state invariant holds for the k + 1 transition. Note that a rule of the form ((I @) (I ,RIGHT)) is never used because the machine never moves left, and by precondition the head starts in position 1. We make an argument for each rule that may be used:

((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule, nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule, nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains `a` and that the head moves to position 2. Therefore, A-INV holds.

((S a) (B ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule, nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains `a` and that the head moves to position 2. Therefore, B-INV holds.

((S b) (C ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule, nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains `b` and that the head moves to position 2. Therefore, C-INV holds.

((A a) (A ,RIGHT)): By inductive hypothesis, A-INV holds. This means that the read part of the input word is a member of $\texttt{a}^*$ and that, the head's position, i $\geq$ 2. Reading the `a` means the read part of the input word continues to be a member of $\texttt{a}^*$ and i $\geq$ 2 continues to hold. Thus, A-INV holds.

((A ,BLANK) (Y ,BLANK)): By inductive hypothesis, A-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*$. Reading a blank means the input word is a member of $\mathtt{a}^*$. Thus, Y-INV holds.

((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*$ and that, the head's position, i $\geq$ 2. Reading the a and moving the head right means the read part of the input word continues to be a member of $\mathtt{a}^*$ and i $\geq$ 2 continues to hold. Thus, B-INV holds.

((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*$ and, the head's position, i $\geq$ 2. Reading a b means the read part of the input word is a member of $\mathtt{a}^*\mathtt{b}$ and that i $\geq$ 2 continues to hold. Thus, C-INV holds.

((C a) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*\mathtt{b}$. Reading an a means the input word is not a member of $\mathtt{a}^*$ nor $\mathtt{a}^*\mathtt{b}$. Thus, N-INV holds.

((C b) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*\mathtt{b}$. Reading an b means the input word is not a member of $\mathtt{a}^*$ nor $\mathtt{a}^*\mathtt{b}$. Thus, N-INV holds.

((C ,BLANK) (Y ,BLANK)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of $\mathtt{a}^*\mathtt{b}$. Reading a blank means the input word is a member of $\mathtt{a}^*\mathtt{b}$. Thus, Y-INV holds. □

Armed with the knowledge that the state invariants always hold, we may proceed to prove the following theorem:

Theorem 4 *L = L(a*Ua*b)*

As before, the proof is divided into two lemmas.

Lemma 3 $w \in L \Leftrightarrow w \in L(a^*Ua^*b)$

Proof
($\Rightarrow$) Assume w $\in$ L. This means that w $\in$ a* or w $\in$ a*b. Given that state invariants always hold, a*Ua*b must halt in Y after reading w. Thus, w $\in$ L(a*Ua*b).

($\Leftarrow$) Assume w $\in$ L(a*Ua*b). This means that a*Ua*b halts in Y after consuming w. Given that the invariants always hold, w $\in$ a* or w $\in$ a*b. Thus, w $\in$ L. □

Lemma 4 *$w \notin L \Leftrightarrow w \notin L(a^*Ua^*b)$*

Proof
($\Rightarrow$) Assume w $\notin$ L. This means that w $\notin$ a* and w $\notin$ a*b. Given that state invariants always hold, a*Ua*b does not halt in Y after reading w. Thus, w $\notin$ L(a*Ua*b).

($\Leftarrow$) Assume w $\notin$ L(a*Ua*b). This means that a*Ua*b does not halt in Y after reading w. Given that state invariants always hold, this means that w $\notin$ a* and w $\notin$ a*b. Thus, w $\notin$ L. $\square$

4 In `a*Ua*b`, states `A` and `B` have the same role. This type of repetition seems rather silly. Design and implement a *deterministic* Turing machine for `L = a*` $\cup$ `a*b`.

5 Let $\Sigma = \{$`a b c`$\}$. Design and implement a nondeterministic Turing machine for `L =` $\{$`w | w is missing at least one element of` $\Sigma\}$

6 Let $\Sigma = \{$`a b`$\}$. Design and implement a nondeterministic Turing machine for `L =` $\{$`w | w has an odd number of a or w has an even number of a and ends with a b`$\}$

62 Turing Machines Decide Regular Languages

After completing the exercises in the previous sections, you may suspect that `tm`s can decide any regular language. If this is true, then `tm`s are at least as powerful as `dfa`s, and we ought to be able to design and implement a constructor for `tm`s that takes as input a `dfa`.

62.1 Design Idea

Every regular language is decided by a `dfa`. We can think of a `dfa` as a `tm` that never moves left and that never writes to the tape. That is, we may think of a `dfa` as a `tm` whose only action is moving right. A `dfa` rule has the following structure:

`(state`$_1$ `symbol state`$_2$`)`

Using such a rule has two effects: it moves the machine from $\texttt{state}_1$ to $\texttt{state}_2$ and moves the input tape's head one space to the right. A Turing machine may simulate such a rule using the following rule:

```
((state1 symbol) (state2 RIGHT))
```

Thus, every rule in the given `dfa` can be converted to a `tm`-rule.

Handling a `dfa`'s final states requires a little care. We cannot simply make every final state in the `dfa` a final state in the constructed `tm`. Recall that a `tm` halts upon reaching a final state. In contrast, a `dfa` may continue to consume input in a final state. We need a mechanism for the constructed `tm` to move to its accepting final state when the `dfa` would be in a final state with no more input to consume. We can observe that based in the `tm`-rule construction above, the constructed `tm`'s head being on the first blank after the input word is equivalent to the `dfa` having read all its input. Therefore, the `tm` needs transition rules from every `dfa` final state on `BLANK` to the `tm`'s final accepting state.

62.2 Implementation

Based on the design idea above, a `dfa` to `tm` constructor is displayed in Fig. 84. A new state for the constructed `tm`'s final accepting state is created using `FSM`'s `generate-symbol`. The `tm`'s states are this new state and the given `dfa`'s states. The input alphabet is that of the given `dfa`. The rules for the `tm` consist of two sets: those that transition into the final accepting state and those that simulate the given `dfa`'s rule. The rules to transition into the final accepting state are constructed using `map`. For every final state in the given `dfa`, a transition into the final accepting state on a blank is created. The rules for simulating `dfa` rules are also created using `map`. For each rule, `(A c B)` a transition is created from `A` on `c` to `B` moving the head right. The `tm`'s start state is the given `dfa`'s start state. The only final state is the generated final state.

Tests are written by using the new constructor to convert a `dfa` into a `tm` and testing that both the original `dfa` and the constructed `tm` yield the same result. In Fig. 84, the `dfa` used for testing is the same `M` defined in Sect. 21.1.

62.3 Correctness

Let `M` be an arbitrary `dfa` and let `T = (dfa2tm M)`. It is not difficult to see that `L(M) = L(T)`. Informally, `T` simulates `M` until `M` would have consumed all its input. At this point, if `M` would have been in a final state, then `T` moves

Fig. 84 Building a `tm` language recognizer from a `dfa`

```
;; DFA for testing
;; L(M) = ab*
(define M (make-dfa `(S F ,DEAD)
                    '(a b)
                    'S
                    '(F)
                    `((S a F)
                      (S b ,DEAD)
                      (F a ,DEAD)
                      (F b F)
                      (,DEAD a ,DEAD)
                      (,DEAD b ,DEAD))
                    'no-dead))

;; dfa → tm
;; Purpose: Build a tm for the language of the given dfa
(define (dfa2tm m)
  (let [(accept-state (generate-symbol 'Y (sm-states m)))]
  (make-tm (cons accept-state (sm-states m))
           (sm-sigma m)
           (append
            (map (λ (f) (list (list f BLANK)
                              (list 'Y BLANK)))
                 (sm-finals m))
            (map (λ (r) (list (list (first r) (second r))
                              (list (third r) RIGHT)))
                 (sm-rules m)))
           (sm-start m)
           (list accept-state)
           accept-state)))

;; Tests for dfa2tm
(define M-tm (dfa2tm M))

(check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
(check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
(check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
(check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
(check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
(check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
(check-equal? (sm-apply M '(a b b b b b))
              (sm-apply M-tm `(,LM a b b b b b)))
```

to its final accepting state. Otherwise, `T` halts in a non-accepting state and rejects.

7 Strengthen the testing suite for `dfa2tm` by transforming more `dfa`s into `tm`s.

8 Mr. Hacker is very upset that the built `tm` has an accepting final state, but does not have any rejecting states explicitly listed as final states. He would like to make the final states:

```
(cons accept-state (sm-states m))
```

His reasoning is that every state in the given `dfa` becomes a reject state because the `tm` only accepts in `accept-state`. What goes wrong if this change is made to the constructor in Fig. 84?

9 Let `M` be an arbitrary `dfa` and let `T = (dfa2tm M)`. Prove that `L(M) = L(T)`.

63 A Turing Machine for $a^n b^n c^n$

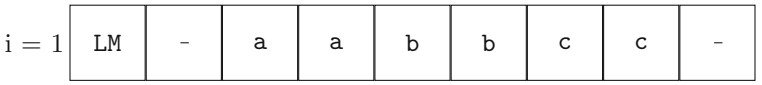
We have established that Turing machines can do anything a `dfa` can do. Are `tm`s more powerful than `dfa`s? Can `tm`s do everything `pda`s can do? Are `tm`s more powerful than `pda`s? The answer to all these questions is affirmative. We shall tackle the problem of proving that `tm`s can do everything `pda`s can do later in this textbook. Our current focus is to establish that `tm`s can perform computations that neither `dfa`s nor `pda`s can perform. To establish this, we design and implement a `tm` to decide a language that is not context-free. The target language is: $L = a^n b^n c^n$.

Designing `tm`s can be a complex process, and the details get messy rather quickly. In a large part, this stems from using mutation to perform computations. It is more difficult to define state invariants when the tape is mutated by the `tm`. A good analogy is traversing a binary tree. It is easier to traverse a binary tree using recursion without mutation than it is to traverse a binary tree using a `while`-loop, a stack, and the mutation of variables. Programming Turing machines is also challenging because they offer a very low level of abstraction. That is, the `API` is much more restricted than any popular higher-level programming language. On the positive side, programming Turing machines provide us with the opportunity to sharpen our programming skills using a given `API`.

63.1 Design Idea

How can a `tm` decide $L = a^n b^n c^n$? Without a doubt, there are many algorithms to solve this problem. As the problem solver, you are free to design

and implement any such algorithm. For our purposes, we shall design and implement an algorithm that traverses the word on the tape multiple times. The precondition is described by the following graphic:

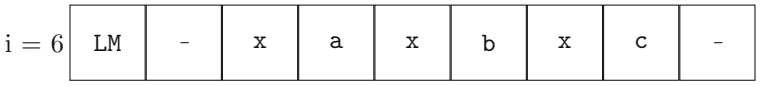


The input word is preceded by a blank, and the head starts on this blank.

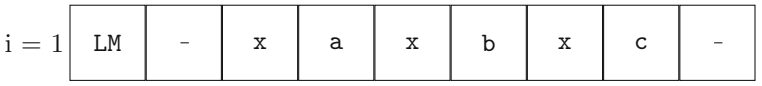
First, the input word is traversed to determine if it is empty or in $\texttt{a}^*\texttt{b}^*\texttt{c}^*$. If it is not, then the machine may halt and reject. If it is empty, then the machine may move to accept. If it is in $\texttt{a}^*\texttt{b}^*\texttt{c}^*$, then the machine must check that there are an equal numbers of as, bs, and cs.

To check that there are an equal number of as, bs, and cs, the input word is traversed and mutated multiple times. If the input word is in L, then at each iteration, a, a b, and a c are substituted with an x. When an a is replaced, the machine nondeterministically decides if it is the last a. If it is not the last a, the machine substitutes the next b and c and loops to substitute the next a. If it is the last a, the machine substitutes the next b and c and moves the head right after the last substituted c. If a blank is read, this means that the machine may move to accept because the mutated word contains only xs and the number of xs is a multiple of 3. If at any point a substitution cannot be made or the element after the last c substituted is not a blank, then the machine halts and rejects because it does not reach the accepting final state.

Let us visualize how this works by continuing with our example. The machine nondeterministically decides that the next a is not the last. During the first traversal, therefore, the first a, b, and c are substituted. The machine ends in a state visualized as follows:



At this point, the machine's head is moved to the blank before the input word to reach this configuration:



Now, the machine nondeterministically decides to substitute the last a. The machine skips the xs to the right (i.e., just one in our example) and substitutes the a leaving the machine in the following state:

i = 3	LM	-	x	x	x	b	x	c	-

The machine proceeds to try to substitute a b. It skips xs to reach and substitute the remaining b. The machine reaches the following state:

i = 5	LM	-	x	x	x	x	x	c	-

The machine proceeds to skip xs to reach and substitute the remaining c, leaving the machine in the following state:

i = 7	LM	-	x	x	x	x	x	x	-

At this point, the machine moves the head to the right to determine if it may accept. This action leaves the machine in the following configuration:

i = 8	LM	-	x	x	x	x	x	x	-

Upon reading the blank, the machine moves to accept.

63.2 Name, Alphabet, and Tests

The machine is named anbncn and the input alphabet Σ = {a b c x}. The machine's precondition is:

tape = `(,LM ,BLANK w) $\wedge$ i = 1, where w$\in${a b c}*

That is, the machine's head starts on the blank before w and w does not contain an x.

The tests are written using input words that are and that are not in L. It is important to test a variety of words that are in not in L. For example, words that are not in a*b*c* and words in a*b*c* that do not have equal number of all three letters are tested. For all tests, care is taken to make sure that the machine's precondition is met. The tests are:

```
(check-equal? (sm-apply anbncn `(,LM ,BLANK b a b c) 1)
              'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a c b) 1)
              'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b c) 1)
              'reject)
(check-equal?
  (sm-apply anbncn `(,LM ,BLANK a a b b b c c) 1)
  'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK c) 1) 'reject)
(check-equal?
  (sm-apply anbncn `(,LM ,BLANK a b c c) 1)
  'reject)
(check-equal?
  (sm-apply anbncn `(,LM ,BLANK a a b b c c a b c) 1)
  'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK) 1) 'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a b c) 1)
              'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b b c c) 1)
              'accept)
(check-equal?
  (sm-apply anbncn `(,LM ,BLANK a a a b b b c c c) 1)
  'accept)
```

63.3 Conditions and States

To determine the states needed it is useful to outline the algorithm that is to be implemented in more detail. For `anbncn`, the algorithm is outlined as follows:

1. The machine starts with the head in position 1 that must be a blank.
2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty, then the machine moves to accept. Otherwise, it determines if the input word is in $\mathtt{a}^+\mathtt{b}^+\mathtt{c}^+$.
3. If the input word is not empty, the machine
   a. tries to read `a`s before a `b`. If a `c` or a blank are read, then the machine halts and rejects.
   b. tries to read `b`s before a `c`. If a blank or an `a` is read, then the machine halts and rejects.
   c. tries to read `c`s before a blank. If an `a` or a `b` is read, then the machine halts and rejects.
   d. The machine moves the head to position 1.

e. The machine skips xs to the right until it reaches an a.
f. The machine substitutes an a and nondeterministically decides to either:
   i. A. Substitute a b.
      B. Substitute a c and loop.
   ii. A. Substitute the last b.
      B. Substitute the last c.
      C. Move the head to the right.
      D. Move to accept if a blank is read.

The algorithm above suggests that 14 states are needed: one for each step outlined and a final accepting state. States are carefully documented to make it easier to implement the needed Turing machine. Let i be the head's position, let w be the input word, and let xs be the xs in w. At the beginning, the head must be on a blank in position 1. The starting state, S, is documented as follows:

```
;; S: i = 1 ∧ tape[i] = BLANK, starting state
```

The machine first determines if the input word is empty. The head's position must be 2, and position 1 of the tape must be a blank. The J state is documented as follows:

```
J: i = 2 AND tape[i-1] = BLANK
```

State A is used to determine that the input word starts with as. This means the head's position must be greater than 2 (i.e., a position beyond the left-end marker and the blank before the input word). As long as as are read, the machine remains in this state. State B is used to determine that as are followed by bs. This means the head's position must be greater than 3. As long as bs are read, the machine remains in this state. State C is used to determine that bs are followed by cs. This means the head's position must be greater than 4. As long as cs are read, the machine remains in C. These states are documented as follows:

```
A: tape[2..i-1] = a⁺ ∧ i > 2
B: tape[2..i-1] = a⁺b⁺ ∧ i > 3
C: tape[2..i-1] = a⁺b⁺c⁺ ∧ i > 4
```

After determining that the input word is in $a^*b^*c^*$, the machine moves the head to position 1. In fact, this must be done every time the leftmost a, b, and c are substituted with an x. This head movement is done in state D and ends when the head reaches the blank before the input word. The input word may have been mutated and has an equal number of as, bs, and cs substituted. Given that the leftmost as, bs, and cs are substituted, there must be an equal number of xs before the first a, the first b, and the first c. In addition, given that the last substitution has not been performed, there must be at least one a, one b, and one c left in the input word. This state is documented as follows:

D: $w = x^n a^+ x^n b^+ x^n c^+ \wedge i \geq 1$

After returning the head to position 1, the machine proceeds to substitute the first a. State E is used for this purpose. The number of as must be greater than 0 because there must be at least one more a substitute. In this state, the substituted as are skipped. That is, the xs before the first a are skipped. This means that the head's position must be greater than 1. Upon reading the first a, the machine nondeterministically decides if it is substituting the next or the last a. We first consider the case in which the machine is not substituting the last a. The machine moves to state F. This means there is one more x for substituted as. In state F, the number of as must be greater than 0, and the number of bs and the number of cs must both be greater than 1 (i.e., the last b and c are not being substituted in this traversal of the input word). The position of the head must be greater than 1 (it may be 1 because it may be the case that the first a has just been substituted). The machine skips the as and xs before the first b. Upon reading the first b, the machine substitutes it and moves to state G. This means there is one more x before the remaining as and before the remaining bs than before the remaining cs. In addition, it means that the number of as and the number of bs are both greater than 0 and the number of cs is greater than 1. The head's position must be greater than 3 because at least an a and a b have been substituted. Upon reading the first c, the machine substitutes it and moves to state D to repeat the process. You may observe that the condition for D is satisfied. These new states are documented as follows:

E: $i > 1 \wedge w = x^n a^+ x^n b^+ x^n c^+$
F: $i > 1 \wedge w = x^{n+1} a^+ x^n bb^+ x^n cc^+$
G: $i > 3 \wedge w = x^{n+1} a^+ x^{n+1} b^+ x^n cc^+$

We now consider substituting the last a, b, and c. After skipping the initial xs and reading the last a in state E, the machine substitutes it and moves to state H to substitute the last b. Thus, the position of the head must be greater than 1. Reaching H means that all as have been substituted with x as well as all bs and cs except the last of each. Recall that the last b and c must be in w. Otherwise, the machine never makes the nondeterministic move to H. This means that w has xs followed by a b followed by xs followed by a c. The length up to the b must be twice the length of what is after it, and the number of xs must be a multiple of 3 plus 1. Upon substituting the last b, the machine moves to a state, I, to substitute the last c. Thus, the position of the head must be greater than 2. In addition, w has xs followed by a c such that the number of xs is a multiple of 3 plus 2. These new states are documented as follows:

H: $i > 1 \wedge w{=}x^+ bx^+ c \wedge |xs| \text{ remainder } 3 = 1 \wedge |x^* b| {=} 2 |x^* c|$
I: $i > 2 \wedge w{=}x^+ c \wedge |xs| \text{ remainder } 3 = 2$

Upon substituting the last c while in state I, the machine moves to, K, a new state. Observe that the head's position must be greater than 3. In K, the x just written is skipped by moving the head to the right, and the machines

moves to, L, a new state to check if a blank is read. Thus, in L, the head's position must be greater than 4. Further observe that in K, the input word is not empty and contains a multiple of 3 number of xs, the read element is an x, and the head position is the length of the mutated input word plus one and that in L, the input word is not empty and contains a multiple of 3 number of xs, and the head position is the length of the mutated input word plus two. The states are documented as follows:

```
K: i > 3 ∧ w = xxxx* ∧ |xs| remainder 3 = 0 ∧ tape[i]=x
   ∧ i=|w|+1
L: i > 4 ∧ w = xxxx* ∧ |xs| remainder 3 = 0 ∧ i=|w|+2
```

Finally, the final accepting state, Y, means that each a, b, and c has been substituted with an x and that the number of xs is divisible by 3. It is documented as follows:

```
Y: w = x* ∧ |xs| remainder 3 = 0, final accepting state
```

63.4 Transition Function, Implementation, and Testing

From the starting state, S, on a blank, the machine moves the head right and moves to J to determine if the input word is empty or is not empty. The only needed transition is:[13]

```
((S ,BLANK) (J ,RIGHT))
```

In state J, the machine determines if the input is empty. If it is empty then, it may move to Y and accept. Otherwise, it moves to A to start the process to determine if the input word is in $a^+b^+c^+$. The needed transition are:

```
((J ,BLANK) (Y ,BLANK))
((J ,BLANK) (A ,RIGHT))
```

The process of determining if the input is in $a^+b^+c^+$ is done using states A, B, and C. In A, reading an a means the head must be moved right. Reading a b means the head is moved to the right, and the machine moves to B to determine that only bs followed by cs remain in the input word. The needed transitions are:

```
((A a) (A ,RIGHT))
((A b) (B ,RIGHT))
```

Recall that the machine is nondeterministic and transition rules from other elements in $\Sigma \cup$ BLANK} are not needed because if encountered, the machine halts in a non-final state and rejects.

[13] We assume all the transition rules are placed inside a quasiquoted list.

In B, reading a b means the head must move to the right. Reading a c means the head is moved right, and the machine moves to C to determine if only cs remain in the input word. The needed transitions are:

```
((B b) (B ,RIGHT))
((B c) (C ,RIGHT))
```

In C, reading a c means the head must move to the right. Reading a blank means the machine transitions to D to move the head left to the blank before the input word. The head may be moved to the left to start the process. The needed transitions are:

```
((C c) (C ,RIGHT))
((C ,BLANK) (D ,LEFT))
```

In D, the machine moves the head left upon reading any element in Σ. Reading a blank means that the machine must transition to E to start the next traversal of the input word and that the head is moved right to start this process. The needed transitions are:

```
((D a) (D ,LEFT))
((D b) (D ,LEFT))
((D c) (D ,LEFT))
((D x) (D ,LEFT))
((D ,BLANK) (E ,RIGHT))
```

The process of mutating the input word is done in states E–I. In E, reading an x means the first a has not been reached and the head must move to the right. Reading an a means the machine must nondeterministically decide if it is or is not the last a. If it is not, the machine mutates the a to an x and moves to F to mutate the first b. If it is, the machine mutates the a to an x and moves to H to mutate the last b. The needed transitions are:

```
((E x) (E ,RIGHT))
((E a) (F x))
((E a) (H x))
```

In F, the machine must skip the x just written to the tape in state E, the remaining as, and the substituted bs. This means moving the head to the right upon reading an a or an x. Reading a b means that it must be substituted with an x, and the machine moves to G to substitute the first c. The needed transition rules are:

```
((F a) (F ,RIGHT))
((F b) (G x))
((F x) (F ,RIGHT))
```

In G, the x just written to the tape in state F, the remaining bs, and the substituted cs must be skipped. For these, the head must be moved to the right. Upon reading a c, it is substituted with an x, and the machine loops back to D to return to the blank before the input word. The needed transition rules are:

```
((G b) (G ,RIGHT))
((G x) (G ,RIGHT))
((G c) (D x))
```

In `H`, the `x` just written in state `E`, and the substituted `b`s must be skipped to substitute the last `b`. This means that upon reading an `x`, the head is moved to the right. Upon reading a `b`, it is substituted, and the machine moves to `I` to substitute the last `c`. The needed transition rules are:

```
((H x) (H ,RIGHT))
((H b) (I x))
```

In `I`, the `x` written in `H` and the substituted `c`s are skipped by moving the head right upon reading an `x`. If a `c` is read, then it is mutated to an `x`, and the machine moves to `K` to start determining if the next element is a blank. In `K`, the head is moved right to skipped the just written `x` in state `I`, and the machine moves to state `L`. Finally, in `L` if a blank is read, the machine moves to accept. The needed transition rules are:

```
((I x) (I ,RIGHT))
((I c) (K x))
((K x) (L ,RIGHT))
((L ,BLANK) (Y ,BLANK)))
```

The implementation of the machine is displayed in Fig. 85. Running the program reveals that all the tests pass. Random testing, using `sm-test`, is of limited use with Turing machines that mutate the tape. Consider, for example, the following tests:

```
> (sm-test anbncn 10)
'(((@ _) accept)
  ((@ c x x x) reject)
  ((@ x c c b a c x c b) reject)
  ((@ x c c b b x b b c) reject)
  ((@ a a a c a c x b) reject)
  ((@ a c x x b a b b b) reject)
  ((@ a b c b b) reject)
  ((@ x c x c x a x) reject)
  ((@ c c a a x a c b a) reject)
  ((@ a c x x a c) reject))
```

The usefulness of random testing is diminished for two reasons. The first, the likeliness of randomly generating a word in $a^n b^n c^n$ is very small. The second, the input word is mutated. This means that there is no way to reconstruct the original input from the provided results. A bit more useful is `sm-showtransitions`. Consider the following trace:

Fig. 85 The Turing machine implementation for $L = a^n b^n c^n$

```
(define anbncn (make-tm '(S A B C D E F G H I J K L Y)
                        '(a b c x)
                        `(((S ,BLANK) (J ,RIGHT))
                          ((J ,BLANK) (Y ,BLANK))
                          ((J a) (A ,RIGHT))
                          ((A a) (A ,RIGHT))
                          ((A b) (B ,RIGHT))
                          ((B b) (B ,RIGHT))
                          ((B c) (C ,RIGHT))
                          ((C c) (C ,RIGHT))
                          ((C ,BLANK) (D ,LEFT))
                          ((D a) (D ,LEFT))
                          ((D b) (D ,LEFT))
                          ((D c) (D ,LEFT))
                          ((D x) (D ,LEFT))
                          ((D ,BLANK) (E ,RIGHT))
                          ((E x) (E ,RIGHT))
                          ((E a) (F x))
                          ((E a) (H x))
                          ((F a) (F ,RIGHT))
                          ((F b) (G x))
                          ((F x) (F ,RIGHT))
                          ((G b) (G ,RIGHT))
                          ((G x) (G ,RIGHT))
                          ((G c) (D x))
                          ((H x) (H ,RIGHT))
                          ((H b) (I x))
                          ((I x) (I ,RIGHT))
                          ((I c) (K x))
                          ((K x) (L ,RIGHT))
                          ((L ,BLANK) (Y ,BLANK)))
                        'S
                        '(Y)
                        'Y))

(check-equal? (sm-apply anbncn `(,LM ,BLANK a a) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK b b b) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK b a b c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a c b) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b b b c c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a b c c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b b c c a b c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK) 1) 'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a b c) 1) 'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b b c c) 1) 'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a a b b b c c c) 1) 'accept)
```

```
> (sm-showtransitions anbncn `(,LM ,BLANK a b c) 1)
'((S 1 (@ _ a b c))
  (J 2 (@ _ a b c))
  (A 3 (@ _ a b c))
  (B 4 (@ _ a b c))
  (C 5 (@ _ a b c _))
  (D 4 (@ _ a b c _))
  (D 3 (@ _ a b c _))
  (D 2 (@ _ a b c _))
  (D 1 (@ _ a b c _))
  (E 2 (@ _ a b c _))
  (H 2 (@ _ x b c _))
  (H 3 (@ _ x b c _))
  (I 3 (@ _ x x c _))
  (I 4 (@ _ x x c _))
  (Y 4 (@ _ x x x _)))
```

The provided trace allows you to observe the movement of the head and the mutations performed to the tape. Thus, you may observe if the desired effects are correctly performed. Remember, however, that such a trace is only provided for input words that lead to accept.

63.5 State Invariant Predicates

To make the implementation of state invariant predicates clearer, an auxiliary function to extract a given word's front symbols that match a given symbol is used. The auxiliary function is:

```
;; word symbol → word
;; Purpose: Return the subword at the front of the
;;          given word that only contains the given
;;          symbol
(define (front-symbs w s)
  (takef w (λ (a) (eq? a s))))

(check-equal? (front-symbs '(a a a c b) 'c) '())
(check-equal? (front-symbs '(a a a c b) 'a) '(a a a))
```

The invariant predicate for `S` determines that the head's position is 1 and a blank is under the head. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that head is in position 1 and
;;          tape[i] = BLANK
(define (S-INV t i)
  (and (= i 1) (eq? (list-ref t i) BLANK)))
```

```
(check-equal? (S-INV `(,LM ,BLANK  a b c) 0) #f)
(check-equal? (S-INV `(,LM a b c c) 1) #f)
(check-equal? (S-INV `(,LM ,BLANK  a b c) 1) #t)
(check-equal? (S-INV `(,LM ,BLANK  a a b b c) 1) #t)
```

The invariant predicate for `A` determines that the head's position is greater than 2 and that tape positions in `[2..i-1]` contain only `a`s. To this end, the input word is extracted from the given tape, and the leading `a`s are extracted from the input word. The predicate is implemented as follows:

```
(define (A-INV t i)
  (and (> i 2)
       (let* [(w (drop (take t i) 2))
              (as (front-symbs w 'a))]
         (equal? as w))))

(check-equal? (A-INV `(,LM ,BLANK  a b c) 0) #f)
(check-equal? (A-INV `(,LM ,BLANK ,BLANK ,BLANK) 2) #f)
(check-equal? (A-INV `(,LM ,BLANK  a b a b c) 4) #f)
(check-equal? (A-INV `(,LM ,BLANK  a b c) 3) #t)
(check-equal? (A-INV `(,LM ,BLANK  a a a b b c) 5) #t)
```

Observe that the second test illustrates that the input word may not be empty.

The invariant predicate for `B` determines that the head's position is greater than 3 and that `tape[2..i-1] = a`$^+$`b`$^+$. To this end, the tape up to position `i - 1` is extracted from the given tape. The left-end marker and blank are dropped to obtain the read part of the input word. The leading `a`s are extracted as well as the `b`s after them. The predicate tests that the read part of the input word is the extracted `a`s followed by the extracted `b`s and that there is at least one of both. It is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine head in position > 2 and
;;          tape[2..i-1] = a+b+
(define (B-INV t i)
  (and (> i 3)
       (let* [(w (drop (take t i) 2))
              (as (front-symbs w 'a))
              (w-as (drop w (length as)))
              (bs (front-symbs w-as 'b))]
         (and (equal? w (append as bs))
              (not (empty? as))
              (not (empty? bs))))))
```

```
(check-equal? (B-INV `(,LM ,BLANK  a b c) 0) #f)
(check-equal? (B-INV `(,LM ,BLANK  a b b c c) 6) #f)
(check-equal? (B-INV `(,LM ,BLANK  a a b b c c) 6) #t)
(check-equal? (B-INV `(,LM ,BLANK  a b b c) 5) #t)
(check-equal? (B-INV `(,LM ,BLANK  a a b b b c c) 7) #t)
```

Observe that the tests allow for the number of as and bs not to be the same.

The invariant predicate for C determines that the head's position is greater than 4 and that tape[2..i-1] = $a^+b^+c^+$. To this end, the tape up to position i is extracted from the given tape. The left-end marker and blank are dropped to obtain the read part of the input word. The leading as are extracted as well as the following bs and the cs after them. The predicate tests that the read part of the input word is the extracted as followed by the extracted bs followed by the extracted cs and that there is at least one of each. It is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine head in position > 3 and
;;          tape[2..i-1]=a+b+c+
(define (C-INV t i)
  (and (> i 4)
       (let* [(w (drop (take t i) 2))
              (as (front-symbs  w 'a))
              (w-as (drop w (length as)))
              (bs (front-symbs  w-as 'b))
              (w-asbs (drop w-as (length bs)))
              (cs (front-symbs w-asbs 'c))]
         (and (equal? w (append as bs cs))
              (not (empty? as))
              (not (empty? bs))
              (not (empty? cs))))))

(check-equal? (C-INV `(,LM ,BLANK  a b b c c) 5) #f)
(check-equal? (C-INV `(,LM ,BLANK  a b b) 4) #f)
(check-equal? (C-INV `(,LM ,BLANK  a b b c c) 6) #t)
(check-equal? (C-INV `(,LM ,BLANK  a b c ,BLANK) 5) #t)
```

The invariant predicates for D–G check a property of i, expect the input word to be in x*a*x*b*x*c*, and check properties of the input word. This leads the predicates to share the following body structure:

```
(and <expression for i property>
     (let* [(w (takef (drop t 2)
                      (λ (s) (not (eq? s BLANK)))))
            (xs1 (front-symbs w 'x))
            (w-xs1 (drop w (length xs1)))
            (as (front-symbs w-xs1 'a))
```

```
              (w-xs1as (drop w-xs1 (length as)))
              (xs2 (front-symbs w-xs1as 'x))
              (w-xs1asxs2 (drop w-xs1as (length xs2)))
              (bs (front-symbs w-xs1asxs2 'b))
              (w-xs1asxs2bs (drop w-xs1asxs2 (length bs)))
              (xs3 (front-symbs w-xs1asxs2bs 'x))
              (w-xs1asbsxs3 (drop w-xs1asxs2bs (length xs3)))
              (cs (front-symbs w-xs1asbsxs3 'c))]
         (and <expressions for input word properties>)))
```

The `let*`-expression extracts the components of the input word. Instead of repeating the local variable definitions in the discussion of each invariant predicate below, the local definitions are denoted by ... . To define each predicate as a function in `FSM` simply substitute ... with the local definitions above.

The invariant predicate for `D` determines that the head's position is greater than or equal to 1 and that the input word is in $x^n a^+ x^n b^+ x^n c^+$, where `n` is a natural number greater than or equal to 1. To do so, it determines that in the input word, there is at least one `a`, one `b`, and `c` and that the number of `x`s before the `a`s equals the number of `x`s before the `b`s and the number of `x`s before the `c`s. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that head position is >= 1 and that
;;          tape[i]=x^n a^+ x^n b^+ x^n c^+
(define (D-INV t i)
  (and (>= i 1)
       (let* [...]
         (and (equal? w (append xs1 as xs2 bs xs3 cs))
              (> (length as) 0)
              (> (length bs) 0)
              (> (length cs) 0)
              (= (length xs1) (length xs2) (length xs3))))))

(check-equal? (D-INV `(,LM ,BLANK b b b c c) 4) #f)
(check-equal? (D-INV `(,LM ,BLANK a b b) 2) #f)
(check-equal? (D-INV `(,LM ,BLANK a b c) 1) #t)
(check-equal? (D-INV `(,LM ,BLANK a a b b c c) 7) #t)
```

The invariant predicate for `E` determines that the head's position is greater than 1 and that the input word is in $x^n a^+ x^n b^+ x^n c^+$, where `n` is a natural number greater than 1. To do so, as done by `D-INV`, it determines that there is at least one `a`, one `b`, and `c` and that the number of `x`s before the `a`s equals the number of `x`s before the `b`s and the number of `x`s before the `c`s. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine head in position > 1 and
;;          input word = x^n a^+ x^n b^+ x^n c^+
(define (E-INV t i)
  (and (> i 1)
       (let* [...]
         (and (equal? w (append xs1 as xs2 bs xs3 cs))
              (> (length as) 0)
              (> (length bs) 0)
              (> (length cs) 0)
              (= (length xs1) (length xs2) (length xs3))))))

(check-equal?
  (E-INV `(,LM ,BLANK x a a x b b x c c ,BLANK) 1)
  #f)
(check-equal?
  (E-INV `(,LM ,BLANK x a a x b c c ,BLANK) 2)
  #f)
(check-equal?
  (E-INV `(,LM ,BLANK x a a x b b x c c ,BLANK) 4)
  #t)
(check-equal? (E-INV `(,LM ,BLANK a a b b c c ,BLANK) 2) #t)
```

The invariant predicate for F determines that the head's position is greater than 1 and that the input word is in $x^{n+1}a^+x^nbb^+x^ncc^+$, where n is a natural number greater than or equal to 1. To this end, it determines that there is at least one a, at least 2 bs, and at least 2 cs, and that the number of xs before the as is 1 more than both the number of xs before the bs and the number of xs before the cs. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine head in position > 1 and
;;          input word = x^{n+1} a^+ x^n bb^+ x^n cc^+
(define (F-INV t i)
  (and (> i 1)
       (let* [...]
         (and (equal? w (append xs1 as xs2 bs xs3 cs))
              (> (length as) 0)
              (> (length bs) 1)
              (> (length cs) 1)
              (= (sub1 (length xs1))
                 (length xs2)
                 (length xs3))))))
```

```
(check-equal?
  (F-INV `(,LM ,BLANK x a a x b b c c c ,BLANK) 3)
  #f)
(check-equal?
  (F-INV `(,LM ,BLANK x a b b c c ,BLANK) 1)
  #f)
(check-equal?
  (F-INV `(,LM ,BLANK x a a b b b c c c ,BLANK) 3)
  #t)
(check-equal?
  (F-INV `(,LM ,BLANK x a b b c c ,BLANK) 2)
  #t)
```

The invariant predicate for `G` determines that the head's position is greater than 3 and that the input word is in $\mathtt{x}^{n+1}\mathtt{a}^{+}\mathtt{x}^{n+1}\mathtt{b}^{+}\mathtt{x}^{n}\mathtt{cc}^{+}$, where `n` is a natural number greater than or equal to 1. To this end, it determines that there is at least one `a`, at least one `b`, and at least two `c`s, that the number of `x`s before the `a`s equals the number of `x`s between the `a`s and the `b`s, and that the number of `x`s before the `a`s is one less than the number of `x`s between the `b`s and the `c`s. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine head in position > 2 and
;;          tape=x^{n+1}a^+x^{n+1}b^+x^ncc^+
(define (G-INV t i)
  (and (> i 3)
       (let* [...]
         (and (equal? w (append xs1 as xs2 bs xs3 cs))
              (> (length as) 0)
              (> (length bs) 0)
              (> (length cs) 1)
              (= (sub1 (length xs1))
                 (sub1 (length xs2))
                 (length xs3))))))

(check-equal?
  (G-INV `(,LM ,BLANK x a a x x b c c c ,BLANK) 3)
  #f)
(check-equal?
  (G-INV `(,LM ,BLANK x a a x b b x c c ,BLANK) 5)
  #f)
(check-equal?
  (G-INV `(,LM ,BLANK x a a x b b c c c ,BLANK) 5)
  #t)
(check-equal?
  (G-INV `(,LM ,BLANK x x a x x b x c c ,BLANK) 8)
  #t)
```

The invariant predicate for H determines that the head's position is greater than 1 and that the input word is in x^+bx^+c. In addition, it determines that the remainder of the number of xs and 3 is 1 and that the length of x^+b is twice the length of x^+c. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine tape[i]=x^+bx^+c and |xs|%3 = 1 and
;;          |x^+b| = 2*|x^+c|
(define (H-INV t i)
  (and (> i 1)
   (let* [(w (drop-right (drop t 2) 1))
          (xs1 (front-symbs w 'x))
          (w-xs1 (drop w (length xs1)))
          (b (front-symbs w-xs1 'b))
          (w-xs1b (drop w-xs1 (length b)))
          (xs2 (front-symbs w-xs1b 'x))
          (w-xs1bxs2 (drop w-xs1b (length xs2)))
          (c (front-symbs w-xs1bxs2 'c))]
     (and (equal? w (append xs1 b xs2 c))
          (= (add1 (length xs1)) (* 2 (add1 (length xs2))))
          (= (length b) 1)
          (= (length c) 1)
          (= (remainder (length (append xs1 xs2)) 3) 1)))))

(check-equal? (H-INV `(,LM ,BLANK a b c ,BLANK) 2) #f)
(check-equal? (H-INV `(,LM ,BLANK x x c ,BLANK) 4) #f)
(check-equal? (H-INV `(,LM ,BLANK x b c ,BLANK) 2) #t)
(check-equal? (H-INV `(,LM ,BLANK x x x b x c ,BLANK) 3) #t)
```

The invariant predicate for I determines that the head's position is greater than 2 and that the input word is in x^+c. In addition, it determines that the remainder of the number of xs and 3 is 2. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine tape[i]=x*c and |xs|%3 = 2
(define (I-INV t i)
  (and (> i 2)
   (let* [(w (drop-right (drop t 2) 1))
          (xs1 (front-symbs w 'x))
          (w-xs1 (drop w (length xs1)))
          (c (front-symbs w-xs1 'c))]
     (and (equal? w (append xs1 c))
          (= (length c) 1)
          (= (remainder (length xs1) 3) 2)))))
```

```
(check-equal? (I-INV `(,LM ,BLANK a b c ,BLANK) 2) #f)
(check-equal? (I-INV `(,LM ,BLANK x c ,BLANK) 3) #f)
(check-equal? (I-INV `(,LM ,BLANK x x c ,BLANK) 4) #t)
(check-equal? (I-INV `(,LM ,BLANK x x x x x c ,BLANK) 7) #t)
```

The invariant predicate for J determines that the head's position is 2 and that at position 1, the tape has a blank. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that head's position is 2 and
;;          tape[1] = BLANK
(define (J-INV tape i)
  (and (= i 2) (eq? (list-ref tape (sub1 i)) BLANK)))

(check-equal? (J-INV `(,LM ,BLANK a b c) 1) #f)
(check-equal? (J-INV `(,LM a a a) 1) #f)
(check-equal? (J-INV `(,LM ,BLANK a b c) 2) #t)
(check-equal? (J-INV `(,LM ,BLANK b b b) 2) #t)
```

K's invariant predicate determines that the input word only contains xs and its length is at least 3 and divisible by 3. In addition, it determines that the head is over the last x. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine if w = xxxx* and |xs|%3 = 0
;;          and tape[i] = x
(define (K-INV t i)
  (let [(w (drop-right (drop t 2) 1))]
    (and (eq? (list-ref t i) 'x) ;;;
         (andmap (λ (s) (eq? s 'x)) w)
         (>= (length w) 3)
         (= (remainder (length w) 3) 0)
         (= i (add1 (length w))))))

(check-equal? (K-INV `(,LM ,BLANK a b c ,BLANK) 3) #f)
(check-equal? (K-INV `(,LM ,BLANK x x c ,BLANK) 3) #f)
(check-equal? (K-INV `(,LM ,BLANK x x x ,BLANK) 4) #t)
(check-equal? (K-INV `(,LM ,BLANK x x x x x x ,BLANK) 7)
              #t)
```

L's invariant predicate determines that the input word only contains xs and its length is at least 3 and divisible by 3. In addition, it determines that the head is over the blank after the last x. The predicate is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine that w = xxxx* and |xs|%3 = 0 and
;;          i = |w| + 2
(define (L-INV t i)
  (let [(w (drop-right (drop t 2) 1))]
    (and (andmap (λ (s) (eq? s 'x)) w)
         (>= (length w) 3)
         (= (remainder (length w) 3) 0)
         (= i (+ (length w) 2)))))

(check-equal? (L-INV `(,LM ,BLANK a ,BLANK) 3) #f)
(check-equal? (L-INV `(,LM ,BLANK x x c ,BLANK) 5) #f)
(check-equal? (L-INV `(,LM ,BLANK x x x ,BLANK) 5) #t)
(check-equal? (L-INV `(,LM ,BLANK x x x x x x ,BLANK) 8) #t)
```

The invariant predicate for `Y` determines that the input word only contains `xs` and that the number of `xs` is divisible by 3. It is implemented as follows:

```
;; tape natnum → Boolean
;; Purpose: Determine input word = x* and |xs|%3 = 0
(define (Y-INV t i)
  (let* [(w (drop-right (drop t 2) 1))]
    (and (andmap (λ (s) (eq? s 'x)) w)
         (= (remainder (length w) 3) 0))))

(check-equal? (Y-INV `(,LM ,BLANK x x c ,BLANK) 3) #f)
(check-equal? (Y-INV `(,LM ,BLANK a b c ,BLANK) 3) #f)
(check-equal? (Y-INV `(,LM ,BLANK ,BLANK) 2) #t)
(check-equal?
  (Y-INV `(,LM ,BLANK x x x x x x ,BLANK) 7)
  #t)
```

Use the invariant predicates in conjunction with the visualization tool to validate that they hold for computations that accept the given word. Make sure you understand why the state invariants hold before proceeding with the correctness argument.

63.6 Correctness

As before, start by proving that the state invariant predicates always hold. Subsequently, we prove that $a^n b^n c^n$ = `L(anbncn)`. We shall use the following definitions:

`L` = $a^n b^n c^n$ `M = anbncn w = input word xs = the xs in w`

63.6.1 Proving State Invariants Hold

Theorem 5 *The state invariants hold when M is applied to w.*

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof

Base case: n = 0
When M starts it is in state S, and by precondition, M's head position is 1, and position 1 is the blank before w. Therefore, S-INV holds.

Inductive Step:
Assume: State invariants hold for a computation of length n = k
Show: State invariants hold for a computation of length n = k + 1

`((S ,BLANK) (J ,RIGHT))`: By inductive hypothesis, S-INV holds. This means M's head position is 1, and a blank is read. Using this transition means the head's position becomes 2, and the tape at position 1 has a blank. Thus, J-INV holds.

`((J ,BLANK) (Y ,BLANK))`: By inductive hypothesis, J-INV holds. This means the head's position is 2, and the tape at position 1 has a blank. Using this rule means that w is empty, and the number of `xs` (i.e., 0) is divisible by 3. Furthermore, the tape is reading a blank. Thus, Y-INV holds.

`((J a) (A ,RIGHT))`: By inductive hypothesis, J-INV holds. This means the head's position is 2, and the tape at position 1 has a blank. Using this rule means that the head is moved to the right to position 3 and that a single `a` and nothing else has been read from w. Thus, A-INV holds.

`((A a) (A ,RIGHT))`: By inductive hypothesis, A-INV holds. This means that the head's position is greater than 2 and that the read part of w only contains `a`s. Using this rule means that the head's position continues to be greater than 2 and that the read part of w continues to only contain `a`s. Thus, A-INV continues to hold.

`((A b) (B ,RIGHT))`: By inductive hypothesis, A-INV holds. This means that the head's position is greater than 2 and that the read part of w only contains `a`s. Using this rule means that the head's position is moved to the right to become greater than 3 and that the read part of w is in $\texttt{a}^+\texttt{b}$. Thus, B-INV holds.

`((B b) (B ,RIGHT))`: By inductive hypothesis, B-INV holds. This means that the head's position is greater than 3 and that w is in $\texttt{a}^+\texttt{b}^+$. Using this

rule means that the head's position continues to be greater than 3 and that the read part of w continues to be in a^+b^+. Thus, B-INV holds.

((B c) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the head's position is greater than 3 and that w is in a^+b^+. Using this rule means that the head's position becomes greater than 4 and that the read part of w is in a^+b^+c. Thus, C-INV holds.

((C c) (C ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the head's position is greater than 4 and that w is in $a^+b^+c^+$. Using this rule means that the head's position continues to be greater than 4 and that the read part of w continues to be in $a^+b^+c^+$. Thus, C-INV holds.

((C ,BLANK) (D ,LEFT)): By inductive hypothesis, C-INV holds. This means that the head's position is greater than 4 and that w is in $a^+b^+c^+$. Using this rule means that the head's position is greater than or equal to 1 and that w is in $x^0a^+x^0b^+x^0c^+$. Thus, D-INV holds.

((D a) (D ,LEFT)): By inductive hypothesis, D-INV holds. This means that the head's position is greater than or equal to 1 and that w is in $x^na^+x^nb^+x^nc^+$, where n is a natural number. Using this rule means that the head has not reached the blank at position 1. Thus, the head's position continues to be greater than or equal to 1. Given that the tape is not mutated, w continues to be in $x^na^+x^nb^+x^nc^+$. Thus, D-INV holds.

((D b) (D ,LEFT)): By inductive hypothesis, D-INV holds. This means that the head's position is greater than or equal to 1 and that w is in $x^na^+x^nb^+x^nc^+$, where n is a natural number. Using this rule means that the head has not reached the blank at position 1. Thus, the head's position continues to be greater than or equal to 1. Given that the tape is not mutated, w continues to be in $x^na^+x^nb^+x^nc^+$. Thus, D-INV holds.

((D c) (D ,LEFT)): By inductive hypothesis, D-INV holds. This means that the head's position is greater than or equal to 1 and that w is in $x^na^+x^nb^+x^nc^+$, where n is a natural number. Using this rule means that the head has not reached the blank at position 1. Thus, the head's position continues to be greater than or equal to 1. Given that the tape is not mutated, w continues to be in $x^na^+x^nb^+x^nc^+$. Thus, D-INV holds.

((D x) (D ,LEFT)): By inductive hypothesis, D-INV holds. This means that the head's position is greater than or equal to 1 and that w is in $x^na^+x^nb^+x^nc^+$, where n is a natural number. Using this rule means that the head has not reached the blank at position 1. Thus, the head's position continues to be greater than or equal to 1. Given that the tape is not mutated, w continues to be in $x^na^+x^nb^+x^nc^+$. Thus, D-INV holds.

((D ,BLANK) (E ,RIGHT)): By inductive hypothesis, D-INV holds. This means that the head's position is greater than or equal to 1 and that w is in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position becomes greater than 1 and, given that the tape is not mutated, that w continues to be in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$. Thus, E-INV holds.

((E x) (E ,RIGHT)): By inductive hypothesis, E-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position continues to be greater than 1 and, given that the tape is not mutated, that w is in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$. Thus, E-INV holds.

((E a) (F x)): By inductive hypothesis, E-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position continues to be greater than 1. In addition, it also means that the last a is not substituted, and therefore, there must be at least two bs and two cs left to substitute. Given that the tape is mutated by substituting the read a with an x, w is in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$. Thus, F-INV holds.

((E a) (H x)): By inductive hypothesis, E-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the last a is substituted. The head's position continues to be greater than 1. Given that the read a is substituted with an x, w is in $\mathtt{x}^{+}\mathtt{b}\mathtt{x}^{+}\mathtt{c}$, such that $|\mathtt{xs}|$ is a multiple of 3 plus 1 and $|x^{+}b| = 2{*}|x^{+}c|$. Thus, H-INV holds.

((F a) (F ,RIGHT)): By inductive hypothesis, F-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position continues to be greater than 1 and, given that the tape is not mutated, w continues to be in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$. Thus, F-INV holds.

((F b) (G x)): By inductive hypothesis, F-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position is greater than 3, because the b substituted is preceded by the left-end marker, a blank, at least one x, and at least one a. In addition, it means that w is in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}+1}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$. Thus, G-INV holds.

((F x) (F ,RIGHT)): By inductive hypothesis, F-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$, where n is a natural number. Using this rule means that the head's position continues to be greater than 1 and, given that the tape is not mutated, w continues to be in $\mathtt{x}^{\mathtt{n}+1}\mathtt{a}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{b}\mathtt{b}^{+}\mathtt{x}^{\mathtt{n}}\mathtt{c}\mathtt{c}^{+}$. Thus, F-INV holds.

((G b) (G ,RIGHT)): By inductive hypothesis, G-INV holds. This means that the head's position is greater than 3 and that w is in $\mathtt{x}^{n+1}\mathtt{a}^+\mathtt{x}^{n+1}\mathtt{b}^+\mathtt{x}^n\mathtt{cc}^+$, where n is a natural number. Using this rule means that the head's position continues to be greater than 3 and, given that the tape is not mutated, w continues to be in $\mathtt{x}^{n+1}\mathtt{a}^+\mathtt{x}^{n+1}\mathtt{b}^+\mathtt{x}^n\mathtt{cc}^+$. Thus, G-INV holds.

((G c) (D x)): By inductive hypothesis, G-INV holds. This means that the head's position is greater than 3 and that w is in $\mathtt{x}^{n+1}\mathtt{a}^+\mathtt{x}^{n+1}\mathtt{b}^+\mathtt{x}^n\mathtt{cc}^+$, where n is a natural number. Using this rule means that the head's position is greater than or equal to 1 and that w is in $\mathtt{x}^n\mathtt{a}^+\mathtt{x}^n\mathtt{b}^+\mathtt{x}^n\mathtt{c}^+$. Thus, D-INV holds.

((H x) (H ,RIGHT)): By inductive hypothesis, H-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^+\mathtt{bx}^+\mathtt{c}$, such that n is a natural number, |xs| is a multiple of 3 plus 1, and $|x^+b| = 2^*|x^+c|$. Using this rule means that the head is moved to the right and w is not mutated. Thus, H-INV holds.

((H b) (I x)): By inductive hypothesis, H-INV holds. This means that the head's position is greater than 1 and that w is in $\mathtt{x}^+\mathtt{bx}^+\mathtt{c}$, such that n is a natural number, |xs| is a multiple of 3 plus 1, and $|x^+b| = 2^*|x^+c|$. Observe that reading a b means that the head's position is greater than 2. Using this rule means that the head's position continues to be greater than 2. In addition, w is mutated by substituting the last b with an x. After this mutation, w is in $\mathtt{x}^+\mathtt{c}$, such that |xs| remainder 3 is 2. Thus, I-INV holds.

((I x) (I ,RIGHT)): By inductive hypothesis, I-INV holds. This means that the head's position is greater than 2 and that w is in $\mathtt{x}^+\mathtt{c}$, such that |xs| remainder 3 is 2. Using this rule means that the head's position is moved to the right without mutating w. Thus, I-INV holds.

((I c) (K x)): By inductive hypothesis, I-INV holds. This means that the head's position is greater than 2 and that w is in $\mathtt{x}^+\mathtt{c}$, such that |xs| remainder 3 is 2. Using this rule means that w is mutated to substitute the last c with an x. That is, w is mutated to be in $\mathtt{xxxx}^*$ such that |xs| is a multiple of 3. In addition, reading a c means that the head's position is greater than 3 (the c is preceded by at least the left-end marker, a blank, and two xs). Finally, using this rules means that the head's over the last x at the |w|+1. Thus, K-INV holds.

((K x) (L ,RIGHT)): By inductive hypothesis, K-INV holds. This means that w is in $\mathtt{xxxx}^*$, the head's position is greater than 3, |xs| is a multiple of 3, the head's position is |w|+1, and the value at the head's position is x. After using this rule, the head's position is greater than 4, w is in $\mathtt{xxxx}^*$, |xs| is a multiple of 3, and the head's position is |w|+2. Thus, L-INV holds.

`((L ,BLANK) (Y ,BLANK))`: By inductive hypothesis, L-INV holds. This means that w is in `xxxx`*, |xs| is a multiple of 3, and the head's position is |w|+2. Using this rule means that the tape's first element after w is a blank, that w is in `x`*, and that |xs| is a multiple of 3. Thus, Y-INV holds. □

63.6.2 Proving `L = L(M)`

As before, the proof is divided into two lemmas. The first is for when `w`∈`L` and the second for when `w`∉`L`

Lemma 5 *w∈L ⇔ w∈L(M)*

Proof
(⇒) Assume w∈L. This means that w = $a^n b^n c^n$. Given that state invariants always hold, there is a computation that has M repeatedly mutating matching `a`s, `b`s, and `c`s to `x` and ending in Y with the input word only having a multiple of 3 number of `x`s. Therefore, w∈L(M).

(⇐) Assume w∈L(M). This means that M halts in Y, the only accepting state, with the input word only having a multiple of 3 number `x`s. Given that the state invariants always hold, the only way for M to reach Y is by the original input being in $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ and substituting equal number of `a`s, `b`s, and `c`s with `x`s. Thus, w∈L. □

Lemma 6 *w∉L ⇔ w∉L(M)*

Proof
(⇒) Assume w∉L. This means that w ≠ $\texttt{a}^n\texttt{b}^n\texttt{c}^n$. Given that state invariants always hold, M cannot halt in Y. Therefore, w∉L(M).

(⇐) Assume w∉L(M). This means that M does not halt in Y. Given that the state invariants always hold, this means that w is not in $\texttt{a}^n\texttt{b}^n\texttt{c}^n$. Thus, w ∉L. □

64 The Turing Tar-Pit

The first recipient of the prestigious Turing Award, Alan Perlis, wrote:

> Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

It refers to the fact that Turing machines are capable of carrying out many computations (like deciding languages that are not context-free) but offer a

very low level of abstraction. That is, Turing machines do not offer any of the common programming constructs that we are accustomed to as computer scientists. For example, they do not offer a looping construct. A consequence of this is that every time we need to loop through (part of) the input word, we must code the loop as part of the transition relation. In this regard, defining Turing machines is akin to programming in assembly. Any program that is written in a higher-level programming language can be written in assembly. We do not, however, commonly program in assembly because higher-level programming languages offer abstractions that make problem-solving easier.

So, why bother studying Turing machines? The answer is twofold. The first, as recurring theme in this book, programming Turing machines provide us with the opportunity to sharpen our skills to program using an `API`. That is, as programmers we must learn to adapt to the programming language that we are asked to use to solve problems. Second, it is a simple model that allows us to easily reason about it and provides a lingua franca for all computer scientists to talk about computation and its limitations (as we shall later in this textbook).

As you practice designing and implementing Turing machines, rely on the design recipe for state machines. The steps of the design recipe assist in managing the design complexity. Also rely on the visualization tool. Test heavily to observe if the state invariants always hold. A state invariant that fails to hold indicates a bug in your design. The bug may be either in the transition relation or in the definition of the predicate. Either way, it indicates that the steps of the design recipe must be revisited.

10 The famous Turing Hacker claims that he has designed and implemented a simpler machine to decide $\mathtt{a^n b^n c^n}$. He claims that there is no need to first check if the input word is in $\mathtt{a^* b^* c^*}$. His proposed Turing machine is displayed in Fig. 86. Is he correct? Carefully justify your answer.

11 Design and implement a Turing machine for $\mathtt{L} = \mathtt{a^n b^n}$. Follow all the steps of the design recipe for state machines.

12 Let Σ `= {a b}`. Design and implement a Turing machine for $\mathtt{L} = \{\mathtt{ww} \mid \mathtt{w} \in \Sigma^*$. Follow all the steps of the design recipe for state machines.

13 Let $\Sigma = \{\mathtt{a\ b\ c}\}$. Design and implement a Turing machine for $\mathtt{L} = \{\mathtt{w} \mid \mathtt{w} \in \Sigma^*\} \wedge$ `w has equal number of as, bs, and cs`. Follow all the steps of the design recipe for state machines.

14 Design and implement a Turing machine for $\mathtt{L} = \mathtt{a^n b^n c^n d^n}$. Follow all the steps of the design recipe for state machines.

Fig. 86 Turing Hacker's proposed machine for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

```
#lang fsm

;; L = a^nb^nc^n
(define anbncn (make-tm '(S D E F G H I J Y)
                        '(a b c x)
                        `(((S ,BLANK) (J ,RIGHT))
                          ((J ,BLANK) (Y ,BLANK))
                          ((J a) (E a))
                          ((D a) (D ,LEFT))
                          ((D b) (D ,LEFT))
                          ((D c) (D ,LEFT))
                          ((D x) (D ,LEFT))
                          ((D ,BLANK) (E ,RIGHT))
                          ((E x) (E ,RIGHT))
                          ((E a) (F x))
                          ((E a) (H x))
                          ((F a) (F ,RIGHT))
                          ((F b) (G x))
                          ((F x) (F ,RIGHT))
                          ((G b) (G ,RIGHT))
                          ((G x) (G ,RIGHT))
                          ((G c) (D x))
                          ((H x) (H ,RIGHT))
                          ((H b) (I x))
                          ((I x) (I ,RIGHT))
                          ((I c) (Y x)))
                        'S
                        '(Y)
                        'Y))

(check-equal? (sm-apply anbncn `(,LM ,BLANK a a) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK b a b c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a c b) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a b b b c c) 1) 'reject)
(check-equal? (sm-apply anbncn `(,LM ,BLANK) 1) 'accept)
(check-equal? (sm-apply anbncn `(,LM ,BLANK a a a b b b c c c) 1) 'accept)
```

15 Let Σ = {a b}. Design and implement a Turing machine for L = {www | w$\in \Sigma^*$}. Follow all the steps of the design recipe for state machines.

16 Let Σ = {a b}. Design and implement a Turing machine for L = {w | w is a palindrome}. Follow all the steps of the design recipe for state machines.

Chapter 16
Turing Machine Composition

Turing machines can do much more than decide languages. They can also perform computations and execute statements. For example, as we saw in the development of `anbncn` in Chap. 15, Turing machines can move to the first blank to the left. As we shall see, Turing machines can also, for example, add numbers, make copies of a word, and compute any number of functions. Before tackling such ambitious goals, however, it is a good idea to design Turing machines that perform simple computations or operations. These simple machines may then be composed to build Turing machines that perform more complex operations. In essence, we shall think of Turing machines as auxiliary functions or subroutines that may be composed to solve problems.

It is possible, of course, to design and implement a Turing machine to solve a problem without using auxiliary Turing machines, much like any program may be written without using auxiliary functions. Such an approach, however, is cumbersome and error-prone and results in machines for which it is too difficult to understand their design. As problem-solvers, to be able to compose Turing machines requires that we clearly specify their precondition and their *postcondition*. A postcondition specifies the state of the machine after it halts. Given two Turing machines, $\mathtt{M}_1$ and $\mathtt{M}_2$, their composition, $\mathtt{M}_1 \circ \mathtt{M}_1$, runs $\mathtt{M}_1$, and when $\mathtt{M}_1$ halts, $\mathtt{M}_2$ runs starting in the configuration the machine is left in by $\mathtt{M}_1$. This configuration must satisfy $\mathtt{M}_2$'s precondition. Otherwise, no claim is made on how the machine behaves. In essence, as problem-solvers, we are responsible for making sure that the precondition of a Turing machine is satisfied before it is allowed to execute.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_16

65 Simple Common Operations

Before composing tms, we shall design and implement Turing machines that perform simple operations that may be used as the building blocks for more complex Turing machines. This is akin to a bottom-up programming approach. Let us assume that Σ = {a b}. For a different alphabet, the machines that perform the same computation are easily designed in the same manner.

65.1 Move Right Machine

To start, consider designing and implementing a machine that only moves its head right one space. The precondition for this machine is that the tape contains an arbitrary value and that the head's position, i, is k, such that k is a nonzero natural number. The postcondition is that the head's position is k+1 and the tape has not been mutated. To achieve this effect the machine transitions to its final state and moves the head right. The implementation is:

```
;;  PRE: tape = (LM w) AND i=k>0, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k+1
(define R (make-tm '(S F)
                   '(a b)
                   `(((S a) (F ,RIGHT))
                     ((S b) (F ,RIGHT))
                     ((S ,BLANK) (F ,RIGHT)))
                   'S
                   '(F)))

(check-equal? (sm-showtransitions R `(,LM a b a) 1)
              `((S 1 (,LM a b a))
                (F 2 (,LM a b a))))
(check-equal? (sm-showtransitions R `(,LM a b a) 3)
              `((S 3 (,LM a b a))
                (F 4 (,LM a b a ,BLANK))))
(check-equal?
  (second (last (sm-showtransitions R `(,LM b b a a) 3)))
  4)
```

It is straightforward to argue that the machine is correct. Therefore, we shall not present a formal argument as suggested by the design recipe for state machines.

65.2 Move Left Machine

Consider designing and implementing a machine that only moves its head left one space. This means that the head's position must be greater than or equal to 1, because the head cannot be moved left if it is on position 0. The implementation is:

```
;;  PRE: tape = (LM w) AND i=k≥1, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k-1
(define L (make-tm '(S H)
                   '(a b)
                   `(((S a) (H ,LEFT))
                     ((S b) (H ,LEFT))
                     ((S ,BLANK) (H ,LEFT)))
                   'S
                   '(H)))

(check-equal? (last (sm-showtransitions L `(,LM a a) 1))
              `(H 0 (,LM a a)))
(check-equal?
  (last (sm-showtransitions L `(,LM ,BLANK b b a) 4))
  `(H 3 (,LM ,BLANK b b a)))
```

Once again, arguing that the machine is correct is straightforward.

65.3 Halt Machine

How do we design a `tm` that starts and immediately halts? This machine must do nothing. It may not read any input, move the head, nor write to the tape. This is achieved by making the starting state a halting state. Given that the machine shall not attempt to apply a transition, the list of transitions may be empty. The halt machine may be implemented as follows:

```
;;  PRE: tape = (LM w), where w in {a b BLANK}*
;; POST: tape = (LM w)
(define HALT (make-tm '(S)
                      '(a b)
                      '()
                      'S
                      '(S)))

(check-equal?
  (last (sm-showtransitions HALT `(,LM ,BLANK) 1))
  `(S 1 (,LM _)))
(check-equal?
  (last (sm-showtransitions  HALT `(,LM a a b a b) 4))
  `(S 4 (,LM a a b a b)))
```

Observe that the tests illustrate that machine does nothing. The starting configuration is the ending configuration.

65.4 Machines That Write to the Tape

Next, consider designing machines that write an element of the alphabet or a blank to the tape. Writing to the tape is done by overwriting an arbitrary tape value and does not require moving the head. As long as the head's position is greater than or equal to 1, the machine may mutate the tape. Therefore, this is specified in the precondition for each of these machines. After starting, each of these machines writes the needed value to the tape regardless of what is read and moves to its halting state. Their implementation is displayed in Fig. 87. Observe that the tests illustrate that the desired effect is achieved.

66 Composing Turing Machines

We now consider designing a Turing machine that performs more than one simple operation. It is useful to reason about such a machine as the composition of the machines for simple common operations. For example, consider designing and implementing a machine that moves the head to the right twice. In the simplest terms, this machine is, `(R ∘ R)`, the composition of `R` with itself. How can this be implemented? We shall follow the steps of the design recipe for state machines. We name the machine `R^2` and Σ `= {a b}`.

66.1 Design Idea

To move the head two positions to the right, the tape may contain an arbitrary value. To simplify the design, we shall require that the head not start in position 0. The pre- and postcondition may be stated as follows:

```
;;  PRE: tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i=k+2 AND w in (a b BLANK)*
```

After starting, the machine moves the head to the right and transitions to an intermediate state representing that one right move has been made. In this intermediate state, the machine moves the head to the right and transitions to its final state.

Fig. 87 Turing machines to write an a, a b, and a blank to the input tape

```
;;  PRE: tape = (LM w) AND i=k>0 AND tape[i]=s, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k AND tape[i]=a
(define Wa (make-tm '(S H)
                    `(a b)
                    `(((S a) (H a))
                      ((S b) (H a))
                      ((S ,BLANK) (H a)))
                    'S
                    '(H)))

(check-equal? (last (sm-showtransitions Wa `(,LM b) 1))
              `(H 1 (,LM a)))
(check-equal? (last (sm-showtransitions Wa `(,LM b a ,BLANK a) 3))
              `(H 3 (,LM b a a a)))

;;  PRE: tape = (LM w) AND i=k>0 AND tape[i]=s, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k AND tape[i]=b
(define Wb (make-tm '(S H)
                    `(a b)
                    `(((S a) (H b))
                      ((S b) (H b))
                      ((S ,BLANK) (H b)))
                    'S
                    '(H)))

(check-equal? (last (sm-showtransitions Wb `(,LM ,BLANK ,BLANK) 2))
              `(H 2 (,LM ,BLANK b)))
(check-equal? (last (sm-showtransitions Wb `(,LM b b b b) 3))
              `(H 3 (,LM b b b b)))

;;  PRE: tape = (LM w) AND i=k>0 AND tape[i]=s, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k AND tape[i]=BLANK
(define WB (make-tm '(S H)
                    `(a b)
                    `(((S a) (H ,BLANK))
                      ((S b) (H ,BLANK))
                      ((S ,BLANK) (H ,BLANK)))
                    'S
                    '(H)))

(check-equal? (last (sm-showtransitions WB `(,LM a a) 1))
              `(H 1 (,LM ,BLANK a)))
(check-equal? (last (sm-showtransitions WB `(,LM a b b b) 3))
              `(H 3 (,LM a b ,BLANK b)))
```

66.2 Tests

The tests must illustrate that the effect has been achieved. That is, the tests must show that the machine's head has moved two positions to the right without mutating the tape. Sample tests are:

```
(check-equal?
  (last (sm-showtransitions R^2 `(,LM a b a) 1))
  `(F 3 (,LM a b a)))
(check-equal?
  (last (sm-showtransitions R^2 `(,LM a b a) 3))
  `(F 5 (,LM a b a ,BLANK ,BLANK)))
(check-equal?
  (last (sm-showtransitions R^2 `(,LM b b a a) 4))
  `(F 6 (,LM b b a a ,BLANK ,BLANK)))
```

Observe that in each test, the head's position ends two positions to the right of the head's position in the initial configuration.

66.3 Transition Function

In the starting state, `S`, the machine moves the head to the right and transitions to the intermediate state `A`, regardless of what is read from the tape. The needed transitions are:

```
((S a) (A ,RIGHT))
((S b) (A ,RIGHT))
((S ,BLANK) (A ,RIGHT))
```

In the intermediate state, `A`, the machine moves the head to the right and transitions to, `F`, the final state regardless of what is read from the tape. The needed transitions are:

```
((A a) (F ,RIGHT))
((A b) (F ,RIGHT))
((A ,BLANK) (F ,RIGHT))
```

66.4 Implementation

The machine's implementation is displayed in Fig. 88. Running the tests reveals that they all pass and give us cautious optimism that the machine is correctly implemented. The development of state invariants and the correctness argument suggested by the design recipe for state machines are omitted, because it is fairly straightforward to see that the machine works.

The important point to realize is that `R^2` is the composition of `R` with itself. Let us compare the transitions from each state:

```
((S a) (A ,RIGHT))          ((A a) (F ,RIGHT))
((S b) (A ,RIGHT))          ((A b) (F ,RIGHT))
((S ,BLANK) (A ,RIGHT))     ((A ,BLANK) (F ,RIGHT))
```

Fig. 88 The Turing machine to move the head two positions to the right

```
;;  PRE: tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i=k+2 AND w in (a b BLANK)*
(define R^2 (make-tm '(S A F)
                     '(a b)
                     `(((S a) (A ,RIGHT))
                       ((S b) (A ,RIGHT))
                       ((S ,BLANK) (A ,RIGHT))
                       ((A a) (F ,RIGHT))
                       ((A b) (F ,RIGHT))
                       ((A ,BLANK) (F ,RIGHT)))
                     'S
                     '(F)))

(check-equal? (last (sm-showtransitions R^2 `(,LM a b a) 1))
              `(F 3 (,LM a b a)))
(check-equal? (last (sm-showtransitions R^2 `(,LM a b a) 3))
              `(F 5 (,LM a b a ,BLANK ,BLANK)))
(check-equal? (last (sm-showtransitions R^2 `(,LM b b a a) 4))
              `(F 6 (,LM b b a a ,BLANK ,BLANK)))
```

Seen side by side like this, it becomes obvious that we have implemented `R` twice in `R^2`. Is this a problem or simply a silly observation? Think about this carefully. Perhaps, for moving the head twice to the right, it is an observation that we are willing to ignore. However, what if we needed to a `tm` that moves the head to the right 20 times? 50 times? Are we to repeatedly implement `R` 20 and 50 times?

By now, you surely realize that moving the head to the right `n` times requires the composition of `R` with itself `n` times. Indeed, as programmers, we have a problem with this. There is no problem-solver who wants to solve the same problem `n` times, and there is no programmer who wants to write the same code `n` times. As programmers, what do we do to avoid writing the same code over and over again? We create an abstraction. That is, we create a function and use the function multiple times instead of writing the same code multiple times. This raises the tantalizing question of whether or not a Turing machine can execute a program that represents a Turing machine.

1 Design and implement a Turing machine that moves its head twice to the left.

2 Design and implement a Turing machine that upon reading an `a` writes a `b` and upon reading a `b` writes an `a`.

3 Design and implement a Turing machine that swaps the read value with the value immediately to the right.

4 Design and implement a Turing machine that swaps the read value with the value immediately to the left.

5 Design and implement a Turing machine that overwrites every `a` with a blank.

67 A Programmed Turing Machine

Our computers can read a program written using a programming language's syntax, read the input for the program, and evaluate the program to compute a desired value or to achieve a desired effect. Are Turing machines powerful enough to do the same? That is, can Turing machines read a program that represents a Turing machine, read the input to the Turing machine represented by the program, and evaluate the given program? It would be truly remarkable, indeed, for a Turing machine to be able to simulate a Turing machine given as input.

Despite their simplicity, Turing machines are powerful enough to be programmed. The universal Turing machine (`UTM`) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. It is given as input (on its tape) a description of the machine to simulate as well as a description of the simulated input tape for the simulated machine. The description of the machine to simulate may be thought of as a program that is interpreted much like your `FSM` programs are interpreted. The `UTM` performs the actions in the description of the machine to simulate on the simulated input tape. To this end, it must track the next action to perform and the position of the head in the simulated input. You may think of the next action to perform as a program counter. It indicates where to continue the execution of the simulated machine once the current action is simulated. The position of the simulated head indicates the next element to be read from the simulated input tape when the current action is executed.

67.1 Moving the Head Right `n` Times

To illustrate that Turing machines are programmable, we shall tackle the problem of moving the head to the right `n` times. That is, we shall create

an abstraction to move the head to the right an arbitrary number of times. To start, we must define how a `tm` that moves its head to the right `n` times is represented. A straightforward representation is `r*ir*`. The `r`s represent the number of times to move the head to the right. That is, the actions of the machine to simulate. You may think of `i` as the program counter. It is placed to the left of the next action to execute. The `r`s before `i` have been executed, and the `r`s after `i` have not yet been executed. The input to the simulated machine is in $\Sigma^*\mathtt{h}\Sigma^*$, where `h` represents the position of the head. The next input element to read is always immediately to the right of `h`. The input to the Turing machine running the program contains `LM`, `BLANK`, the representation of the machine to simulate, `BLANK`, and the simulated input tape for the simulated machine. For instance, consider the following input tape:

```
`(,LM ,BLANK i r r r ,BLANK h a b a)
```

The `tm` simulated moves the head to the right 3 times (indicated by the 3 `r`s). Given that `i` is before the first `r`, none of these movements have been performed. The input word to the simulated machine is `(a b a)`, and the simulated head is over the first `a`. In contrast, consider the following input tape:

```
`(,LM ,BLANK r i r r r ,BLANK b b a h b ,BLANK)
```

The `tm` simulated moves the head to the right 4 times. Given that `i` is before the second `r`, the first of these movements has been performed. The input word to the simulated machine is `(b b a b)`, and the simulated head is over the last `b`. Since the simulated head has been moved to the right once, we can conclude it started over the `a`.

How does the Turing machine evaluate our program work? It works much like the von Neumann architecture. The Turing machine fetches the next action to perform, moves the program counter (i.e., `i`), performs the action, and loops. The loop stops when there are no more instructions to perform (e.g., `i` is at the position containing the blank between the simulated machine's description and the simulated input tape). In essence, the `tm` evaluating the program implements a fetch-execute cycle similar to what you may have studied in a computer architecture course.

67.2 Design Idea

The machine being simulated does not require complex updating of the program counter (i.e., no types of jumps required). Updating the program counter is simply done by moving `i` to the right. For more complex prob-

lems (as we shall see), jumps are required. Luckily, for this problem, we may implement a jump-free version `tm`. The pre- and postcondition are stated as follows:

```
;;  PRE: tape = (LM BLANK i p BLANK w h v) AND
;;       head on first blank in position 1
;; POST: tape = (LM BLANK p i BLANK o) OR
;;              (LM BLANK p i BLANK u)
;;       AND head on second blank,
;; where p = r^n for n∈ℕ,
;;       o = w′hv′ where |w′|-|w| = n,
;;       u = w′v′BLANK*h where |w′v′BLANK*|-|w| = n,
;;       w,w′,v,v′∈Σ*, wv = w′v′
```

The precondition states that the simulated program counter, `i`, starts before the first action in, `p`, the simulated `tm`'s description, and the simulated head is anywhere immediately before, in between, or after, `wv`, the simulated machine's input. The postcondition states that `p` and `wv` are not mutated and that `h` has been moved `n` spaces to the right.

The machine operates as follows:

1. Starts with the head on the blank in position 1 of the tape
2. Attempts to find the `i` to the right (i.e., the next instruction to execute)
3. If done executing the program, the `tm` running the program moves to halt. Otherwise, it moves to the next step.
4. Move `i` to the right (i.e., to the next instruction to execute, if any, after the current instruction)
5. Find `h`
6. Execute current instruction (i.e., move `h` to the right)
7. Find `i`
8. Loop to step 3

We shall implement a transition function for the above steps. Note that the `tm` shall not be able to simulate an arbitrary `tm`. Our `tm` shall only be capable of evaluating programs (i.e., Turing machine descriptions) that perform a single operation an arbitrary number of times: move the head to the right. As we shall see, `tm`s are capable of evaluating an arbitrary `tm`. For our illustrative purposes, the current simplification suffices. As computer scientists, we know that a program written in a higher-level programming language is represented by compiled code, akin to a `tm` description, which is evaluated by a computer, akin to a `tm`, to produce a result or an effect.

We shall assume that the input to the simulated machine only has `a`s and `b`s. Therefore, the alphabet for the `tm` is: {`a b h i r`}. Note that the simulated machine operates on a tape that only contains `a`s, `b`s, and blanks. Only the interpreting `tm` makes use of `i`, `r`, and `h`.

67.3 Tests

The tests are written to simulate `tm`s that move the head right 0 or more times regardless of the length of the simulated input word. In addition, the tests illustrate that the simulated machines move the head right the desired number of times regardless of where the head starts in the simulated input. For example, tests are written with the simulated head before, in the middle, and at the end of the simulated word. A sample set of tests is:

```
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i ,BLANK h a a)
                           1))
 `(Y 3 (,LM ,BLANK i ,BLANK h a a)))
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i r r ,BLANK a h b a)
                           1))
 `(Y 5 (,LM ,BLANK r r i ,BLANK a b a h)))
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i r r r ,BLANK h a b a)
                           1))
 `(Y 6 (,LM ,BLANK r r r i ,BLANK a b a h)))
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i r r ,BLANK a h a b a)
                           1))
 `(Y 5 (,LM ,BLANK r r i ,BLANK a a b h a)))
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i r r r ,BLANK h)
                           1))
 `(Y 6 (,LM ,BLANK r r r i ,BLANK ,BLANK ,BLANK ,BLANK h)))
(check-equal?
 (last (sm-showtransitions PR^N
                           `(,LM ,BLANK i r ,BLANK a h)
                           1))
 `(Y 4 (,LM ,BLANK r i ,BLANK a ,BLANK h)))
```

67.4 Transitions

From the start state, `S`, the machine moves right to the first instruction if any. This is done using an intermediate step to skip over, the program counter, `i`. The needed transitions are:

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

In state `Q` the machine decides if it is done evaluating the program. If it reads a blank, the machine moves to, `Y`, the final state. Otherwise, it swaps the next instruction (the read `r`) with `i` in the previous position. This swap sets the machine to execute the next instruction, if any, after the current instruction is executed. After the swap, the machine moves to state, `B`, to search for `h`. The needed transitions are:

```
((Q ,BLANK) (Y ,BLANK))
((Q r) (M i))
((M i) (N ,LEFT))
((N i) (B r))
```

In state `B`, the machine moves right until it finds `h`. When `h` is found, the machine moves to `D` and moves the head right to the element the simulated machine reads. The needed transitions are:

```
((B r) (B ,RIGHT))
((B i) (B ,RIGHT))
((B ,BLANK) (B ,RIGHT))
((B a) (B ,RIGHT))
((B b) (B ,RIGHT))
((B h) (D ,RIGHT))
```

In state `D`, the machine executes the next instruction. That is, it swaps `h` with the element read by the simulated machine. For each element, the simulated machine may read a different intermediate state is needed to perform the swap. The intermediate state is used to remember the element read by the simulated machine. For example, `E` is used to remember that an `a` was read by the simulated machine. The `tm` writes `h` to the tape and transitions to the appropriate intermediate state. In an intermediate state, the `tm` moves its head left and transitions to another intermediate state to overwrite `h` with the appropriate symbol. For instance, from `E`, the machine transitions to `F` to write `a` to complete the swap of `h` and `a`. After the swap is complete, the

`tm` moves to a state, `K`, to fetch the next instruction. The needed transitions are:

```
((D a) (E h))
((D b) (G h))
((D ,BLANK) (I h))
((E h) (F ,LEFT))
((F h) (K a))
((G h) (H ,LEFT))
((H h) (K b))
((I h) (J ,LEFT))
((J h) (K ,BLANK))
```

In state `K`, the `tm` moves left until it finds `i`. When `i` is read, the `tm` moves right and transitions to `Q` to execute, if any, the next instruction. The needed transitions are:

```
((K a) (K ,LEFT))
((K b) (K ,LEFT))
((K ,BLANK) (K ,LEFT))
((K r) (K ,LEFT))
((K i) (Q ,RIGHT))
```

67.5 Implementation

The complete implementation of programmable `tm` to move `n` spaces to the right is displayed in Fig. 89. Running the tests reveals that they all pass. As illustrated by the tests, the `tm` takes as input a `tm` description along with a description of its input tape and successfully moves the simulated machine's head to the right the number of times specified in the given `tm` description. We no longer need to implement `R` multiple times. All we need to do is provide the proper `tm` and input tape descriptions to move the simulated head to the right a specified number of times.

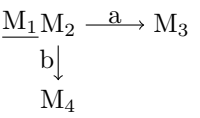
Clearly, it is still rather difficult to program even the simplest tasks using a `tm`. This is said even though we have not looked at problems that require conditional and unconditional jumping. If programming solutions to simple problems is too difficult, then in all likelihood, we need a better abstraction. That is, we need a programming language that is conducive to making it easier to express the solution to a problem, that is, a programming language that makes it easier to compose Turing machines. If you are convinced that `tm`s are programmable, then let's move to the next section to make the composition of Turing machines easier. We shall study a domain-specific language within `FSM` to compose `tm`s that supports conditional jumps, unconditional jumps, and abstraction over values (i.e., variables).

Fig. 89 A programmable `tm` for moving `n` spaces to the right

```
(define PR^N (make-tm '(S A B D E F G H I J K M N Q Y)
                      '(a b h i r)
                      `(((S ,BLANK) (A ,RIGHT))
                        ((A i) (Q ,RIGHT))
                        ((Q ,BLANK) (Y ,BLANK))
                        ((Q r) (M i))
                        ((M i) (N ,LEFT))
                        ((N i) (B r))
                        ((B r) (B ,RIGHT))
                        ((B i) (B ,RIGHT))
                        ((B ,BLANK) (B ,RIGHT))
                        ((B a) (B ,RIGHT))
                        ((B b) (B ,RIGHT))
                        ((B h) (D ,RIGHT))
                        ((D a) (E h))
                        ((D b) (G h))
                        ((D ,BLANK) (I h))
                        ((E h) (F ,LEFT))
                        ((F h) (K a))
                        ((G h) (H ,LEFT))
                        ((H h) (K b))
                        ((I h) (J ,LEFT))
                        ((J h) (K ,BLANK))
                        ((K a) (K ,LEFT))
                        ((K b) (K ,LEFT))
                        ((K ,BLANK) (K ,LEFT))
                        ((K r) (K ,LEFT))
                        ((K i) (Q ,RIGHT)))
                      'S
                      '(Y)))

(check-equal? (last (sm-showtransitions PR^N `(,LM ,BLANK i ,BLANK h a a) 1))
              `(Y 3 (,LM ,BLANK i ,BLANK h a a)))
(check-equal?
  (last (sm-showtransitions PR^N `(,LM ,BLANK i r r ,BLANK a h b a) 1))
  `(Y 5 (,LM ,BLANK r r i ,BLANK a b a h)))
(check-equal?
  (last (sm-showtransitions PR^N `(,LM ,BLANK i r r r ,BLANK h a b a) 1))
  `(Y 6 (,LM ,BLANK r r r i ,BLANK a b a h)))
(check-equal?
  (last (sm-showtransitions PR^N `(,LM ,BLANK i r r ,BLANK a h a b a) 1))
  `(Y 5 (,LM ,BLANK r r i ,BLANK a a b h a)))
(check-equal?
  (last (sm-showtransitions PR^N `(,LM ,BLANK i r r r ,BLANK h) 1))
  `(Y 6 (,LM ,BLANK r r r i ,BLANK ,BLANK ,BLANK ,BLANK h)))
(check-equal? (last (sm-showtransitions PR^N `(,LM ,BLANK i r ,BLANK a h) 1))
              `(Y 4 (,LM ,BLANK r i ,BLANK a ,BLANK h)))
```

68 The Universal Turing Machine

Composed Turing machines may be visualized as a graphic that resembles a flow chart. Consider, for example, the following graphic:



`M1--4` are descriptions of Turing machines. For our current purposes, their purpose is irrelevant. The graphic is a description of a Turing machine that starts by executing `M1` (denoted by being underlined). When `M1` halts, `M2` is executed starting in the machine's configuration after `M1` halts. When `M2` halts, a decision is made. If an `a` is read, then `M3` is executed, and the machine halts, given that there is nothing more to execute. If a `b` is read, then `M4` is executed, and the machine halts for the same reason. Observe that in this example, a conditional jump is required to either execute `M3` or `M4`.

We shall implement the `tm` composition specified by such graphics using a domain-specific programming language that is likely to remind you of programming in assembly with conditional jumps, `GOTO`s, and variables. The domain-specific language is used to write `tm` descriptions. That is, programs that represent a Turing machine. Given a composed `tm` description (`cmtd`), the constructor `combine-tms` is used to build a composed `tm` (`ctm`). This constructor takes as input a `ctmd` and the alphabet for the machine defined by the given `ctmd`. It returns a `ctm`. There is one observer, `ctm-run`, that takes as input a `ctm`, the input tape for the given `ctm`, and the initial position of the head on the given input tape. The observer `ctm-run` is `FSM`'s implementation of what is known as the *universal Turing machine* (`UTM`). The `UTM` simulates the machine described by the given `ctmd` using the given tape and the given starting head position. It returns the machine's configuration when the given `ctm` halts.

68.1 Syntax

A `ctmd` is a list defined as follows:

1. empty list
2. (cons `m` `ctmd`), where `m` is either a `tm` or a `ctmd`
3. (cons LABEL `ctmd`)
4. (cons (list GOTO LABEL) `ctmd`)
5. (cons (BRANCH (listof (list symbol (list GOTO LABEL)))) `ctmd`)
6. (cons ((VAR symbol) `ctmd`) `ctmd`)
7. (cons variable `ctmd`)

The empty `ctmd` represents a machine that must only halt. It is equivalent to `HALT` defined in Sect. 65.3. If during the execution of a `ctm` the empty list is encountered, then execution halts, and the existing `tm` configuration is returned. The second variety states that a `ctmd` may be a list that contains a machine, `m`, and `ctmd`. The machine, `m`, may be a `tm` or a `ctmd`. The execution is this variety of `ctmd` first executes `m` and then continues with the rest of the `ctmd`.

The third variety introduces a label to a `ctmd`. The label is a number used for unconditional branching. It represents the start of a `ctmd` that must be executed when control moves to this point in the `ctmd`. The fourth variety introduces a `GOTO` statement to a `ctmd`. This is how unconditional branching occurs. A `GOTO` statement contains a label that is jumped to when it is executed. That is, control is passed to the `ctmd` designated by the label.

The fifth variety introduces a conditional branching point to a `ctmd`. A `BRANCH` statement contains a list of branching options. Each option is a list that contains a symbol and an unconditional jump. If the read element matches the symbol in a branching option, then control is passed to the `ctmd` specified by the label in the unconditional jump. One of the symbols must match the element being read. Otherwise, an error is thrown.

The sixth variety introduces an abstraction. A `VAR` statement contains a symbol representing a variable and an embedded `ctmd`. The variable is introduced to capture the value being read on the tape. The scope of the variable is the embedded `ctmd`. In the embedded `ctmd`, the seventh `ctmd` variety may occur. When variable is encountered, the input tape is mutated. The value of the variable is written at the head's position.

68.2 Design Principles

When a `ctmd` is visualized as a graphic, then we may more easily design its implementation. For example, the composition of two machines may be represented as the sequence of their names. Consider implementing a `cmtd` that moves the head right twice. Graphically, we may represented as:

R R

Neither jumping nor variables are needed for the implementation. The `ctmd` only needs to sequence `R`s twice. The needed `ctmd` is implemented as follows:

```
;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b)*
;; POST: tape = (LM w) AND i=k+2 AND w in (a b)*
(define RR (combine-tms (list R R) '(a b)))

(check-equal? (ctm-run RR `(,LM b a a) 1)
              `(F 3 (,LM b a a)))
(check-equal? (ctm-run RR `(,LM a a a) 2)
              `(F 4 (,LM a a a ,BLANK)))
```

```
(check-equal? (ctm-run RR `(,LM a b b a) 3)
              `(F 5 (,LM a b b a ,BLANK)))
(check-equal? (ctm-run RR `(,LM b) 1)
              `(F 3 (,LM b ,BLANK ,BLANK)))
```

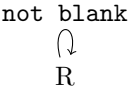
Observe how much easier the implementation becomes with the proper abstraction. That is, using the `UTM` does, indeed, make it easier to program simple tasks. Do not, however, let this lead you to believe that programming `tm`s is easy. It still may require attention to many low-level details akin to programming in assembly.

To simplify the graphics used to illustrate `ctm`s, superscripts are used to abbreviate the graphic when a machine is repeatedly composed with itself. For example, the following abbreviated graphics denote the `ctm`s to move the head twice and thrice to the right:

$$\mathrm{R}^2$$

$$\mathrm{R}^3$$

If there is an incoming arrow into a machine, then a label is needed. If a machine has one or more outgoing labeled arrows, then a branch statement is needed. For instance, consider designing a machine that moves to the first blank to the right. This machine must move to the right and then examine the tape. If a blank is read, it ought to halt. Otherwise, it must loop and repeat the process. Graphically, we may describe this `ctm` as follows:



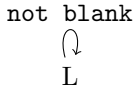
What is this graphic communicating? It is stating that the `UTM` executes `R`. When `R` halts, the tape is examined. If a blank is read, the machine halts. Otherwise, the machine branches (back) to `R`. A label is used to transfer control back to `R`. The `ctm` is implemented as follows:

```
;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i>k AND tape[i] = BLANK
;;       AND tape[k+1..i-1] ≠ BLANK
(define FBR (combine-tms
              (list 0
                    R
                    (cons BRANCH
                          (list (list 'a (list GOTO 0))
                                (list 'b (list GOTO 0))
                                (list BLANK (list GOTO 10))))
                    10)
             (list 'a 'b)))
```

```
(check-equal? (ctm-run FBR `(,LM ,BLANK) 1)
              `(F 2 (,LM ,BLANK ,BLANK)))
(check-equal? (ctm-run FBR `(,LM a a b b b ,BLANK a a) 2)
              `(F 6 (,LM a a b b b ,BLANK a a)))
(check-equal? (ctm-run FBR `(,LM ,BLANK ,BLANK b b) 3)
              `(F 5 (,LM ,BLANK ,BLANK b b ,BLANK)))
```

The label 0 is used to identify where `R` is in the `ctm`. After `R`, there is a `BRANCH` statement with three branches. The blank branch jumps to label 10 to an empty `ctm`. This means that the machine halts after taking this branch. Otherwise, in all other branches, the machine unconditionally jumps to label 0 to continue with the `ctm`'s execution. There is no need for a branch to process reading `LM`, because the machine never reads `LM` from its starting head position and to the right. The tests illustrate that the proper effect is achieved: the head moves to the first blank after its initial position.

A problem that is deceptively similar to finding the first blank to the right is finding the first blank to the left. This machine must move to the left and then examine the tape. If a blank is read, it ought to halt. Otherwise, it must loop and repeat the process. Graphically, we may describe this `ctm` as follows:



Consider making part of the precondition the head's position, `i`, a value `k>0`. The machine must move left at least once. Therefore, we may decide to state in the postcondition that `i<k`. Under this design, the implementation becomes:

```
;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i<k AND tape[i]=BLANK
;;       AND tape[i+1..|w|] != BLANK
(define FBL (combine-tms
              (list 0
                    L
                    (cons BRANCH
                          (list (list 'a (list GOTO 0))
                                (list 'b (list GOTO 0))
                                (list BLANK (list GOTO 1))
                                (list LM (list GOTO 0))))
                   1)
              (list 'a 'b)))

(check-equal? (ctm-run FBL `(,LM ,BLANK a a b) 4)
              `(H 1 (,LM ,BLANK a a b)))
(check-equal?
  (ctm-run FBL `(,LM a ,BLANK a b ,BLANK b a b b) 8)
  `(H 5 (,LM a ,BLANK a b ,BLANK b a b b)))
```

Observe that when a blank is read, after moving left, the machine jumps to label 1 and halts. The machine halts because the `ctmd` at label 1 is empty. The `BRANCH` statement in this case has a branch for processing `LM`, because `LM` may be read as it moves left. Running the program reveals that the tests pass. We must, however, be careful about concluding that the design is correct. Consider running `FBL` as follows:

```
(ctm-run FBL `(,LM a a b) 3)
```

The machine moves left onto the left-end marker. Therefore, it loops and tries to move left again. This time, however, `L` must move right, because it reads the left-end marker. `FBL` then moves left after looping again. This means that the head ends in position 0 reading the left-end marker. Thus, the machine goes into an infinite recursion. This underlines the importance of design and proper documentation. `FBL`'s precondition needs to be strengthened to state that there must be a blank to the left of the head's position:

```
;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;;       AND there exists j<i such that tape[j]=BLANK
```

In this manner, running `FBL` as above is an improper use of the machine. Any programmer using `FBL` now knows that they are responsible for only using `FBL` when there is at least one blank to the left of the head's position.

Finally, consider designing a `ctm` that reads the current element off the tape and overwrites the two positions to the right with the read element. Given that a `tm` cannot write the left-end marker to the tape, the head's position must be greater than 0. The machine must remember the read value in order to mutate the two positions to the right. This is an instance when a variable is needed. In the `ctmd`, we declare a variable to capture the value read. Then, in the variable's scope, we may have a `ctmd` that moves the head right, writes the value of the variable, moves the head right again, writes the value of the variable again, and halts. Before the machine starts, the value on the input tape is arbitrary and `i` is `k`. When the machine halts, `i` is `k+2`, and positions `k-2`, `k-1`, and `k` are the same, and the rest of the values on the tape remain unchanged. The machine may be documented and implemented as follows:

```
;;PRE: tape = (LM a_1 ... a_k a_k+1 a_k+2 a_k+3 ... a_n-1)
;;      ∧ i=k>0 ∧ a_j∈{a b BLANK}
;;POST: tape = (LM a_1 ... a_k a_k a_k a_k+3 ... a_n-1)
;;       ∧ i=k+2
(define WTWICER (combine-tms `(((VAR s) ,R s ,R s))
                             (list 'a 'b)))
```

```
(check-equal? (ctm-run WTWICER `(,LM a ,BLANK) 1)
              `(h 3 (,LM a a a)))
(check-equal? (ctm-run WTWICER `(,LM b a a a ,BLANK) 1)
              `(h 3 (,LM b b b a _)))
(check-equal?
  (ctm-run WTWICER `(,LM a b ,BLANK ,BLANK a) 2)
  `(h 4 (,LM a b b b a)))
```

Running the tests reveals that they all pass.

A precautionary word is now advised. Programming the `UTM` may be very difficult, and it is fairly easy and frequent to have bugs in the machine designed. Given that the `UTM` does not support a debugger nor print statements, you will have to rely on your design skills to debug your machines. This is part of the Turing tar-pit.

6 Let Σ = {`a b`}. Design and implement a `ctm` to move the head to the second blank to the right.

7 Let Σ = {`a b`}. Design and implement a `ctm` to move the head to the second blank to the left.

8 Let Σ = {`a b`}. Design and implement a `ctm` to swap the value read by the head with the value to its right.

9 Let Σ = {`a b`}. Design and implement a `ctm` to erase a word that has a blank before and after it.

10 Let Σ = {`a b`}. Design and implement a `ctm` that overwrites the first blank to the right with the read value.

69 Computing with Turing Machines

As has been suggested before, `tm`s can do more than decide languages, move the input tape head, and mutate the tape. For example, we have seen that `tm`s can perform searches. Evidence of this are `FBR` and `FBL` that search for a blank, respectively, to the right and to the left. In addition to searching, `tm`s can also compute the value of functions. A `tm`, `M`, that computes the value of a function, `f`, reads the needed values from the input tape and writes the value of `f` to the tape. Such `tm`s may be implemented using `make-tm` or `combine-tms`. Which should be used? Unless otherwise instructed, use whichever you feel makes it easier to perform the computation.

69.1 f(a b) = a + b

Consider the problem of adding two natural numbers. This problem sounds deceptively simple. After all, we all know how to add two natural numbers. How did we learn to add two numbers? First, we were taught about the representation of natural numbers. Second, we were taught about the rules for adding natural numbers based on the chosen representation. For instance, in grade school, we were taught how to represent numbers using the digits `0--9` (i.e., decimal notation), and then we were taught the rules adding two numbers represented using decimals. As a computer science student, it is unlikely that grade school was the only place you learned to add two natural numbers. In a computer architecture course, in all likelihood, you were taught how to represent integers using two complement notation and then were taught how to add two integers represented using this notation.

What is the connection with developing `tm` to add two natural numbers? There are two things we must define before any such `tm` may be implemented. First, we must define how to represent natural numbers. Second, we must define how to add two natural numbers represented using the chosen notation.

69.1.1 Design Idea

First, we must choose a representation for natural numbers. Two clear candidates are decimal and two's complement notation. This would require defining the sum of two digits or two bits and defining how a carry is propagated. Both certainly plausible approaches, but we shall explore a representation that simplifies the addition operation: unary notation. In unary notation, there is a single digit, say `d`, used to represent natural numbers. A natural number is defined as follows:

```
A natural number in unary notation (nn) is either:
  1. (BLANK)
  2. (d nn)
```

The first subtype represents 0 that we are all familiar with in decimal notation. The second subtype is used to represent a nonzero natural number. For instance, 0, 3, and 8 are represented as follows:

```
(BLANK)     (d d d)     (d d d d d d d d)
```

Given two `nn` separated on a tape by a blank, addition becomes straightforward:

1. Mutate the blank between the two given natural numbers to d
2. Mutate the last i in the second argument, if any, to a blank

For example, consider the following input:

```
(LM BLANK d d BLANK d d d)
```

The first argument is 2, and the second argument is 3. Following the two steps above yields:

```
(LM BLANK d d d d d)
```

This is the representation of 5, which is the sum of 2 and 3.

To implement and make the machine useful to others, we must specify the pre- and postcondition. Let `a` and `b` be two natural numbers represented using unary notation, and let `i` represent the position of the head. We shall say that the input tape contains the left-end marker, a blank, `a`, a blank, and `b` (followed, of course, by an infinite number of blanks). The head starts at the blank before `a`. When the machine halts, the input tape ought to contain the left-end marker, a blank, and $\texttt{d}^{\texttt{numd(a)+numd(b)}}$, where `numd(x)` is the number of `d` in `x`. In addition, the head ought to be on the first blank after the result. The head's position after the sum is computed depending on whether or not either `a` or `b` represent 0. If both `a` and `b` represent 0, then the head ought to be in position 3. That is, the tape contains the left-end marker, a blank before the result, and a blank for the result, and the head is on the next blank after the result. If either `a` or `b` do not represent 0, then the head ought to be in position equal to the number of `d`s in `a` and `b` plus 2. That is, the head is on the first blank after the result. The pre- and postcondition may be written as follows:

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2    otherwise,
;;       where numd(x) = the number of ds in x
```

How does the machine mutate the tape to get from the state described in the precondition to the state described in the postcondition? When the machine starts, the machine is in the state described by the precondition. It then performs the following abstract steps:

1. Skip `a` and write `d` in blank after `a`
2. Skip `b` and move head left
3. Mutate the position under the head to a blank
4. Move the head left and decide if the result is empty:
   a. If a `d` is read move right and halt
   b. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt

These steps suggest eight states are needed: the starting state, the final state, and six additional states for the steps above. Each step above requires a state except moving the head right twice. Moving the head right twice requires two

states: one to move to the result (i.e., the blank representing 0) and one to move to the blank after the result.

69.1.2 State Documentation

The starting state, `S`, represents the precondition. It is documented as follows:

```
S: i = 1 and tape = (LM BLANK a BLANK b)
```

State `A` is used to skip `a`. This means that the head's position must always be less than or equal to the number of `d`s in `a` plus two. Two is added to the number of `d`s in `a` to account for the tape's first two positions containing the left-end marker and the blank before `a`. This state is documented as follows:

```
A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

State `B` is used to skip `b` after mutating the blank between `a` and `b`. This means that the head's position is always less than or equal to the number of `d`s in `a` and `b` plus 3. The three account for the left-end marker, the blank at the beginning of the tape, as well as the `d` that substituted the blank between `a` and `b`. `B` is documented as follows:

```
B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
```

State `C` is used to mutate the last `d` on the tape to a blank. This means that `i` must equal the number of `d`s in `a` and `b` plus 2. This state is documented as follows:

```
C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
```

After mutating the last `d` on the tape, the head has not moved, and the written part of the tape ends with `a` followed by `b`. That is, we may think of `b` as absorbing the `d` between `a` and `b` to replace its last `d` that was mutated to a blank. This state is captured by `D`:

```
D: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
```

From `D`, the head is moved left to determine if the output is empty. When the head is moved, the machine transitions to `E` that is documented as follows:

```
E: i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
```

In `E`, the head is moved right, and the machine must decide what state to transition to. If the output is 0 (i.e., a `BLANK` is read), then the machine transitions to `G` to move the head to the right again before halting. Otherwise, the head is moved to the right, and the machine transitions to `F` to halt. State `G` is documented as follows:

```
G: i = numd(a) + numd(b) + 2 and tape = (LM BLANK)
```

Finally, state F represents the postcondition. It is documented as follows:

```
F: i = 3                          if a = b = 0
     = numd(a) + numd(b) + 2      otherwise
   and tape = (LM BLANK a b)
```

69.1.3 Transitions

In S, the machine only needs to move the head right and transition to A to start the process of skipping a. The only transition needed is:

```
((S ,BLANK) (A ,RIGHT))
```

In A, the machine needs to skip all ds by moving the head right. When a blank is read, the tape is mutated to make it a d, and the machine transitions to B to skip b. The needed transitions are:

```
((A d) (A ,RIGHT))
((A ,BLANK) (B d))
```

In A, the machine needs to skip all ds by moving the head right. When a blank is read, the machine moves the head one space to the left to position it over the last d on the tape and transitions to C. The needed transitions are:

```
((B d) (B ,RIGHT))
((B ,BLANK) (C ,LEFT))
```

In C, the read d is mutated to a blank, and the machine transitions to D. The only transition needed is:

```
((C d) (D ,BLANK))
```

In D, the machine moves the head left and transitions to E to decide if the output is 0. The needed transition is:

```
((D ,BLANK) (E ,LEFT))
```

In E, the machine decides if the output is 0. The output is 0 if it reads a blank. In this case, it moves the head right and transitions to G to make one more move right. If the output is not 0, then the machine moves the head right (i.e., the blank after the result) and transitions to F to halt. The needed transitions are:

```
((E d) (F ,RIGHT))
((E ,BLANK) (G ,RIGHT))
```

In G, the machine reads the blank representing the output of 0. It moves the head to the right (i.e., the blank after the result) and transitions to F to halt. The needed transition is:

```
((G ,BLANK) (F ,RIGHT))
```

Fig. 90 A Turing machine to add two natural numbers

```
;; numd(x) = the number of ds in x
;; State documentation
;; S: i = 1 and tape = (LM BLANK a BLANK b)
;; A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
;; B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
;; C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
;; D: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
;; E: i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
;; F: i = 3                      if a = b = 0
;;      = numd(a) + numd(b) + 2 otherwise
;;    and tape = (LM BLANK a b)
;; G: i = numd(a) + numd(b) + 2 and tape = (LM BLANK)

;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                      if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                       ((C d) (D ,BLANK))
                       ((D ,BLANK) (E ,LEFT))
                       ((E d) (F ,RIGHT))
                       ((E ,BLANK) (G ,RIGHT))
                       ((G ,BLANK) (F ,RIGHT)))
                     'S
                     '(F)))

(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

69.1.4 Implementation

The implementation of a `tm` to add two natural numbers is displayed in Fig. 90. Running the tests reveals that they all pass.

69.1.5 Correctness

Computations that perform mutations are more difficult to prove correct. The reason for this is that any values that originally existed on the tape are lost after mutating them. This makes it impossible, for example, to infer previous tape values. For instance, consider the tape value after `ADD` transitions into `B`. A possible value is:

```
(LM BLANK d d d d d d)
```

The blank between `a` and `b` has been mutated to a `d`. This makes it impossible, for example, to discern the value of `a` and the value of `b`. That is, we cannot determine the number of `d`s in `a` and the number of `d`s in `b`. It is for this reason that we cannot write invariant predicates that require extracting the values of `a` and `b` from the tape. In general, mutation makes it impossible to extract needed values from the tape.

To establish the correctness of mutation-based computations performed by a `tm`, we may use triples consisting of an assertion, a transition rule, and an assertion. The first assertion is the precondition for the transition rule. It states what must be true before using the transition rule. That is, what must be true in the from-state. The second assertion is the postcondition for the transition rule. It states what must be true after using the transition rule. That is, what is true for the to-state. Concretely, consider an arbitrary transition rule:

```
((X i) (Y a))
```

The corresponding triple is:

```
<Assertion about X's role>
((X i) (Y a))
<Assertion about Y's role>
```

The assertions simply state that the invariant predicates hold. That is, the assertions are based on the role each state plays. A triple is valid if assuming the precondition and the execution of the transition implies the postcondition.

This approach to machine correctness is akin to using Hoare Logic to establish the correctness of a mutation-based program written in a higher-level programming language. The proof of partial correctness is done, as before, by induction on the number of transitions performed.

Theorem 1 *ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.*

Proof
Proof by induction on the number of transitions performed by a computation.

Base case (n = 0):

The machine starts in S, and by assumption, the precondition is true. Therefore, the role of S is satisfied.

Inductive Step:
Assume: State roles satisfied for n = k
Show: that State roles satisfied for n = k + 1

We establish a valid triple for each transition rule. In each triple, the precondition is the assertion for the state the machine is in. The postcondition is the assertion for the state the machine transitions to. By inductive hypothesis, the precondition in each triple holds.

```
i = 1 and tape = (LM BLANK a BLANK b)
((S ,BLANK) (A ,RIGHT))
i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

The machine moves the head right and transitions to A. Given that the head starts in position 1 and the head moves right, the head ends in position 2. It is certainly the case that 2 is less than or equal to numd(a) + 2. The tape remains unchanged. Therefore, the postcondition holds.

```
i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
((A d) (A ,RIGHT))
i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

In `A`, reading an `a` means that the blank in between `a` and `b` is to the right of the head. That is, i < numd(a) + 2. The transition moves the head to the right and does not mutate the tape. Thus, the postcondition holds.

```
i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
((A ,BLANK) (B d))
i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
```

In `A`, reading a blank means that i = numd(a) + 2 < numd(a) + numd(b) + 3. That is, it is reading the blank in between `a` and `b`. The transition mutates the read blank to `d` and moves to `B`. This means after using the transition i ≤ numd(a) + numd(b) + 3 and tape = (LM BLANK a d b). Thus, the postcondition holds.

```
i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
((B d) (B ,RIGHT))
i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
```

In `B`, reading a `d` means i < numd(a) + numd(b) + 3. The transition moves the head to the right and does not mutate the tape. Thus, the postcondition holds.

```
i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
((B ,BLANK) (C ,LEFT))
i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
```

In `B`, reading a blank means i = numd(a) + numd(b) + 3. The transition moves the head to the left and does not mutate the tape. Thus, the postcondition holds.

```
i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
((C d) (D ,BLANK))
i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
```

In `C`, reading a `d` means the head on the last `d` on the tape. Observe that tape contains numd(a) + numd(b) + 1 `d`s. The transition does not move the head and mutates the read `d` to a blank. This mutation means that `a d b` becomes `a b`. Thus, the postcondition holds.

```
i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
((D ,BLANK) (E ,LEFT))
i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
```

In `D`, reading a blank means the blank after `a b` is read. The transition moves the head to the left and does not mutate the tape. Thus, the postcondition holds.

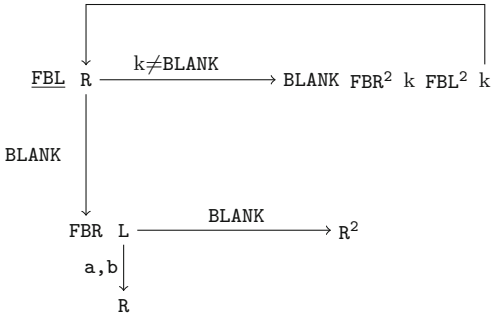
```
i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
((E d) (F ,RIGHT))
i = 3                          if a = b = 0
  = numd(a) + numd(b) + 2 otherwise
∧ tape = (LM BLANK a b)
```

In `E`, reading a `d` means `a b` does not represent 0 and the head is on the last `d` of `a b`. The transition moves the head right and does not mutate the tape. Thus, the postcondition holds.

```
i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
((E ,BLANK) (G ,RIGHT))
i = numd(a) + numd(b) + 2 and tape = (LM BLANK BLANK)
```

In `E`, reading a blank means that `a b` represents 0 and that the head is on the first blank on the tape. Observe that the tape is `(LM BLANK)`. The transition moves the head to the right and does not mutate the tape. Thus, the postcondition holds.

Fig. 91 A ctm diagram for copy(w) = w BLANK w




```
i = numd(a) + numd(b) + 2 and tape = (LM BLANK BLANK)
((G ,BLANK) (F ,RIGHT))
i = 3                          if a = b = 0
  = numd(a) + numd(b) + 2 otherwise
∧ tape = (LM BLANK a b)
```

In G, the input tape is (LM) = (LM BLANK BLANK), and the head is on the second blank. This implies that a and b both represent 0 and, of course, their sum is 0. Therefore, the head is reading the blank representing the result. The transition moves the head to the right and does not mutate the tape. Thus, the postcondition holds. □

The proof establishes that if ADD's precondition is met, then, after ADD executes, the sum of the two given natural numbers is left on the tape with the head on the blank after the sum.

69.2 copy(w) = w BLANK w

The copy function takes as input a word w$\in \Sigma^*$ and returns two copies of w separated by a blank. The first step in designing a machine to compute copy is to specify the configuration when the machine starts and the configuration when the machine halts. That is, we must clearly state the pre- and postconditions. The choices are many, and you are free to choose among them. We shall design a machine with the following specification:

```
;;  PRE: tape = (LM BLANK w BLANK) and head on blank after w
;; POST: tape = (LM BLANK w BLANK w BLANK) and
                head on blank after second w
```

Observe that w's copy starts immediately after the second blank.

69.2.1 Design Idea

We may think of `w` as consisting of `n` symbols. That is, $\texttt{w} = \texttt{a}_0 \ldots \texttt{a}_{\texttt{n}-1}$, where $\texttt{n} \geq 0$. If `n` is 0, then `w` is the empty word (i.e., just a blank). The goal is to copy each $\texttt{a}_\texttt{i}$ to the second blank to the right starting with $\texttt{a}_0$. We copy to the second blank to the right because there must be a blank between `w`'s two copies.

The machine starts by moving to the first blank to the left and then moving right. At this point, the machine either reads the next element to copy or reads the blank after `w`. The machine must branch based on the element read. At this first branching point, if a blank is read, then the tape contains two copies of `w` separated by a blank, and the machine must place the head correctly to satisfy the postcondition. The head must be moved to the first blank to the right and then moved left. A second branch is needed to determine if `w` is empty. If a blank is read, then `w` is empty, and the head is moved twice to the right to satisfy the postcondition. Otherwise, the head is moved once to the right to satisfy the postcondition.

If an alphabet element is read at the first branching point, then the machine must make a copy of the element and loop to repeat the process for any remaining elements to copy. The read value is captured in a variable and then overwritten with a blank. This blank is a placeholder to remember the read element's position. The machine moves to the second blank to the right and mutates this blank to be the value of the variable. It then moves to the second blank to the left and restores the value previously overwritten with a blank. The machine loops to move right and reach the first branching point again.

Figure 91 displays the `ctm` diagram for the copy machine outlined above. Observe that we have already implemented all the auxiliary Turing machines in Sects. 65 and 68.2. Therefore, all we need to do is define the `ctm`. The `ctm`'s starting point is denoted by the underlined machine (i.e., `FBL`). Six labels are needed: one for each arrow and one to mark the end of the machine. This last labeled is jumped to when the machine ought to halt.

69.2.2 Implementation

The machine is named `COPY` and Σ `= {a b}`. The `ctmd` implementation is displayed in Fig. 92.

The tests use the value returned by `ctm-run`, but omit the state. The state is omitted because an auxiliary `tm`'s halting state is arbitrarily named. The important features are the head's position and the contents of the tape. Therefore, only the rest of the result returned by `ctm-run` is tested. The tests in Fig. 92 cover both the empty word and nonempty words. The first test is for the empty word denoted by a blank. Observe that the precondition is met by having the head start on the third blank on an empty tape. The

Fig. 92 The `ctmd` for the copy machine

```
;;  PRE: tape = (LM BLANK w BLANK) and head on blank after w
;; POST: tape = (LM BLANK w BLANK w BLANK) and head on blank after second w
(define COPY (combine-tms
              (list FBL
                    0
                      R
                      (cons BRANCH (list (list BLANK (list GOTO 2))
                                         (list 'a (list GOTO 1))
                                         (list 'b (list GOTO 1))))
                    1
                      (list (list VAR 'k)
                            WB
                            FBR
                            FBR
                            'k
                            FBL
                            FBL
                            'k
                            (list GOTO 0))
                    2
                      FBR
                      L
                      (cons BRANCH (list (list BLANK (list GOTO 3))
                                         (list 'a (list GOTO 4))
                                         (list 'b (list GOTO 4))))
                    3
                      RR
                      (list GOTO 5)
                    4
                      R
                      (list GOTO 5)
                    5)
              `(a b)))

(check-equal? (rest (ctm-run COPY `(,LM ,BLANK ,BLANK ,BLANK) 3))
              `(5 (,LM  ,BLANK ,BLANK ,BLANK ,BLANK ,BLANK)))
(check-equal? (rest (ctm-run COPY `(,LM ,BLANK a b b ,BLANK) 5))
              `(9 (,LM ,BLANK a b b ,BLANK a b b ,BLANK)))
(check-equal? (rest (ctm-run COPY `(,LM ,BLANK b b b b ,BLANK)) 5)
              `(11 (,LM ,BLANK b b b b ,BLANK b b b b ,BLANK)))
```

expected result has the head on the fifth blank on an empty tape. Observe that the postcondition is satisfied, because the tape is: `(LM BLANK w BLANK w BLANK)`. The tests for nonempty words are easier to understand. For each, the precondition is met by having the head on the blank after the word, and the postcondition is met by having the head on the blank after the copy of the given word.

The `ctmd` is written based on the `ctm` diagram in Fig. 91. The machine starts by simulating `FBL`. Label 0 is used to mark the beginning of the machine that decides if all elements of the given word have been copied. It starts by simulating `R` and then branching. If a blank is read, the `ctm` goes to label 2 that marks the beginning of the machine that places the head to meet the postcondition. Otherwise, the `ctm` goes to label 1 that marks the beginning of the machine that copies the next element.

At label 1, the variable `k` is declared to capture the read value. The machine then simulates `WB`, `FBR` twice, writes the value of `k` to the tape, simulates `FBL` twice, writes the value of `k` to the tape, and unconditionally jumps to label 0.

At label 2, the machine simulates `FBR`, `L`, and branches to properly place the head to satisfy the postcondition. If a blank is read, the machine branches to label 3. Otherwise, it branches to label 4.

At labels 3 and 4, respectively, the machine moves right twice and once. This properly places the head to satisfy the postcondition. In both cases, the machine unconditionally jumps to label 5 and halts.

69.2.3 Correctness

To argue the correctness of the machine, we shall use Hoare triples as done in Sect. 69.1.5. We assume that the auxiliary `tm`s work correctly. That is, every auxiliary `tm` satisfies its postcondition when the precondition is met.

When `COPY` starts, the given word may be empty or nonempty. If it is empty, it is denoted by a blank. If it is nonempty, the given word is denoted by $\mathtt{a}_1 \ldots \mathtt{a}_n$. By precondition, the tape contains the left-end marker, a blank, and the input word. The head is on the first blank after the input word. Observe that the precondition for `FBL` holds. Therefore, the triple for the first machine action is:

$$\begin{array}{l} (LM\ BLANK\ a_1 \ldots a_n\ \underline{BLANK}) \ \vee\ (LM\ BLANK\ BLANK\ \underline{BLANK}) \\ \mathtt{FBL} \\ (LM\ \underline{BLANK}\ a_1 \ldots a_n\ BLANK) \ \vee\ (LM\ BLANK\ \underline{BLANK}) \end{array}$$

If the input word is nonempty, then the head is moved to the blank before it. Otherwise, the head is moved over the blank that denotes the input word.

Label 0 may be reached by one of two ways: after the initial `FBL` or by unconditional branch after copying an element. This means that before `R` is executed, the machine may be in one of three configurations. If label 0 is reached from the initial `FBL`, then the configuration may be one of the configurations in the postcondition in the triple above. If label 0 is reached after copying $\mathtt{a}_\mathtt{i}$, then the head is over $\mathtt{a}_\mathtt{i}$. After `R` is executed, the head is either over the blank after the empty input word or over the next element to copy. Without loss of generality, regardless of its position on the tape, in

the post-assertion, the next element to copy is denoted as $\mathtt{a_j}$. The resulting triple is:

(LM $\underline{BLANK}$ $a_1 \ldots a_n$ *BLANK)* $\vee$
(LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\vee$ *(LM BLANK* $\underline{BLANK}$*)*
R
(LM BLANK $a_1 \ldots \underline{a_j} \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK* $\underline{BLANK}$*)*

Label 1 is only reached if there is another element to copy. The index used to refer to this element is arbitrary, and for ease of integration with the rest of the loop, it is once again denoted $\mathtt{a_i}$ in the triple. The first action is to introduce the variable `k` to capture $\mathtt{a_i}$'s value. Therefore, the resulting triple is:

(LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
`(list VAR 'k)`
(LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$

The next action is to move the head to the first blank to the right. This places the head on the blank after the input word. The action and the postcondition are:

FBR
(LM BLANK $a_1 \ldots a_i \ldots a_n$ $\underline{BLANK}$ $a_1 \ldots a_{i-1}\}$*)* $\wedge$ $k{=}a_i$

The next action moves the head to the first blank to the right. This places the head on the blank after the copied elements. The action and the postcondition are:

FBR
(LM BLANK $a_1 \ldots BLANK \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ $\underline{BLANK}$*)* $\wedge$ $k{=}a_i$

The next action mutates the tape. It changes the blank read to $\mathtt{a_i}$. The action and postcondition are:

`'k`
(LM BLANK $a_1 \ldots BLANK \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ $\underline{a_i}$*)* $\wedge$ $k{=}a_i$

The next action moves the head to the first blank to the left placing it over the blank after the input word. The action and postcondition are:

FBL
(LM BLANK $a_1 \ldots BLANK \ldots a_n$ $\underline{BLANK}$ $a_1 \ldots a_{i-1}$ a_i*)* $\wedge$ $k{=}a_i$

The next action moves the head to the first blank to the left placing it over the blank that substituted $\mathtt{a_i}$. The action and postcondition are:

FBL
(LM BLANK $a_1 \ldots \underline{BLANK} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* $\wedge$ $k{=}a_i$

The next action mutates the read blank to be $\mathtt{a_i}$. The action and postcondition are:

```
'k
(LM BLANK a1...ai...an BLANK a1...ai−1 ai)
```

Observe that $\mathtt{a_i}$ has been successfully copied and the machine may safely loop to label 0 because the precondition for label 0 holds.

Label 2 is only reached by reading a blank in the branch of the machine at label 0. Thus, the postcondition for label 0 is the precondition for label 2. The first action is to move the head to the first blank to the right. Regardless of the input word variety, the head is placed over the blank after the input word's copy. The resulting triple is:

```
(LM BLANK a1...an BLANK a1...an) ∨
(LM BLANK BLANK BLANK)
FBR
(LM BLANK a1...an BLANK a1...an BLANK) ∨
(LM BLANK BLANK BLANK BLANK BLANK)
```

The next action moves the head left. This places the head on the copy's $\mathtt{a_n}$ if the input word is not empty. Otherwise, it places the head over the input word's copy. The action and postcondition are:

```
L
(LM BLANK a1...an BLANK a1...an BLANK) ∨
(LM BLANK BLANK BLANK BLANK BLANK)
```

Label 3 is only reached if a blank is read at the end of the machine at label 2. This means the input word is empty, and the head is over it. The action moves the head right twice. The action and postcondition are:

```
(LM BLANK BLANK BLANK BLANK BLANK)
RR
(LM BLANK BLANK BLANK BLANK BLANK BLANK)
```

Observe that the configuration in the second assertion satisfies `COPY`'s postcondition. Therefore, the machine may unconditionally jump to label 5 and halt.

Label 4 is only reached if the head is over the last element of the nonempty input word's copy at the end of label 2. The action moves the head right. The action and postcondition are:

```
(LM BLANK a1...an BLANK a1...an BLANK)
R
(LM BLANK a1...an BLANK a1...an BLANK)
```

Observe that the configuration in the second assertion satisfies `COPY`'s postcondition. Therefore, the machine may unconditionally jump to label 5 and halt.

This completes the correctness argument for COPY. If COPY's precondition holds, then the machine only halts when the postcondition holds.

11 Simplify the design of COPY by implementing and using machines to move to the second blank to the right and move to the second blank to the left. Update the correctness proof to take into account the use of these machines.

12 Design and implement a Turing machine to compute f(a b) = a - b. Assume a > b. Provide a correctness proof.

13 Let Σ = {a b}. Design and implement a Turing machine to compute rev(w) = w^R. That is, a tm that reverses its input. Provide a correctness proof.

14 Let Σ = {a b}. Design and implement a ctm that shifts its input word one space to the right. Provide a correctness proof.

15 Let Σ = {a b}. Design and implement a ctm that shifts its input word one space to the left. Provide a correctness proof.

16 Design and implement a ctm to compute mult(a b) = a * b. That is, a machine that multiplies two natural numbers. Provide a correctness proof.

Chapter 17
Turing Machine Extensions

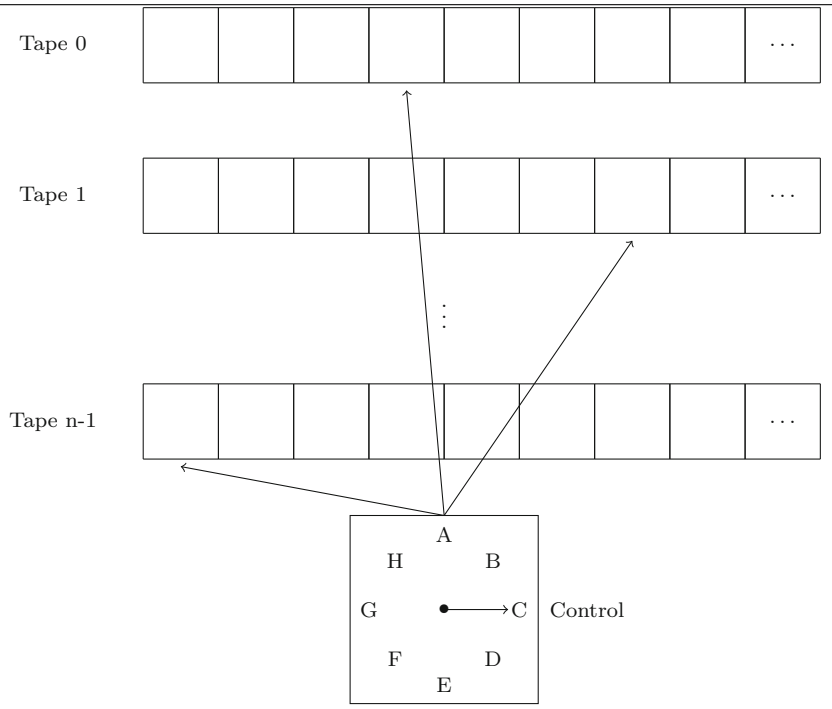
We have seen that Turing machines are powerful enough to recognize languages that are not context-free and to compute arbitrary functions. They are hard to program, because they do not offer the abstractions readily available in modern higher-level programming languages. Nonetheless, this does not diminish their computational power.

To date, an unanswered question in computer science is whether or not there are (undiscovered) machines that are more powerful. There have been attempts to strengthen Turing machines akin to strengthening `ndfa`'s with a stack to create, computation-wise, a more powerful machine. These attempts have not yielded a machine that can recognize a language nor compute a function that a standard Turing machine cannot recognize or compute. It is, indeed, a testament to the Turing machine's computational power. Each of the proposed strengthened `tm`s can be simulated by a standard `tm`. Many computer scientists today believe that the Turing machine is the most powerful computational device and there are no more powerful automata.

A natural question to ask is: Why study new automata models that are not more powerful than the Turing machine? The answer is convenience. The additional features added in attempts to strengthen the Turing machine can make problem-solving easier. Therefore, we are free to use these features knowing that their use may be eliminated. That is, any solution using a "strengthened" Turing machine may be simulated by a standard Turing machine.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_17

Fig. 93 A graphical representation of the multitape Turing machine



70 The Multitape Turing Machine

A proposed extension endows a Turing machine with multiple tapes. A graphical representation is depicted in Fig. 93. Instead of a single infinite tape, the machine has `n` infinite tapes. There is a separate head for each tape and a control module that tracks the machine's state. In one step, the machine reads the value under each head. Based on the values read and the current machine state, each head either mutates their tape, moves right, or moves left and the machine transitions to a state. A standard Turing machine is, therefore, a multitape Turing machine with a single tape.

More formally, a multitape Turing machine (`mttm`) with `n`-tapes is an instance of:

```
(make-mttm K Σ S F δ n [Y])
```

The inputs to the constructor are defined as follows:

K: A list of states. Each state is denoted by a symbol that represents a capital letter in the Roman alphabet.

Σ: A list of symbols or digits. Each symbol represents a lowercase letter in the Roman alphabet.

S: The starting state. It must be a member of K.
F: A list of final states. Each state must be a member of K.
δ: A list of transition rules defining a transition relation. Each transition rule has the following type:

```
(list (list state (list symbol^n))
      (list state (list action^n)))
```

The first tuple contains the state the machine is in and a list of all the elements read by the n heads (the first element corresponds to what is read by the head on tape 0 and the last element corresponds to what is read by the head on tape n-1). The second tuple contains the state the machine transitions into and the n actions taken by the heads (the first element corresponds to the action taken by the head on tape 0 and the last element corresponds to the action taken by the head on tape n-1).
n: A natural number greater than or equal to 1 representing the number of tapes.
Y: An optional argument to define a language recognizer. It represents the accepting state and must be a member of K.

A computation for a mttm, M, is denoted by a list of configurations that M traverses during a computation. A configuration is denoted by a list on length n + 1, where n is the number of tapes. The first element is the machine's state. The rest of the elements are lists that contain a head position and a tape's contents. The first corresponds to tape 0 and the last to tape n - 1. For instance, consider the following configuration:

```
`(D
  (9 (,LM ,BLANK a a b b c c d d ,BLANK))
  (3 (,BLANK b b ,BLANK))
  (3 (,BLANK c c ,BLANK))
  (3 (,BLANK d d ,BLANK)))
```

The machine is in state D. On tape 0 the head is on position 9, and on tapes 1–3, the heads are on position 3.

A transition made by the machine is denoted using $\vdash$. $C_i \vdash C_j$ is valid for M if and only if M can move from C_i to C_j using a single transition. Zero or more moves by M is denoted using $\vdash^*$. $C_i \vdash^* C_j$ is valid for M if and only if M can move from C_i to C_j using zero or more transitions. Finally, an mttm language recognizer accepts a word, w, if there is a computation that reaches, Y, its accepting state.

We adopt the convention that an mttm's input is provided on tape 0 (whose first element is the left-end marker), and the initial position of the head on tape 0 may be specified when the machine is applied. This is akin to a standard tm. The other tapes are initially blank and do not contain the left-end marker, and their heads start in position 0 (i.e., the initial blank). An mttm's output, if any, is provided on tape 0 when it halts, and the contents of the other tapes are ignored.

71 L = {w | w Has Equal Number of as, bs, and cs}

We proceed to design and implement an `mttm` to decide the language of all words that have an equal number of `a`s, `b`s, and `c`s. We follow the design recipe for state machines.

71.1 Name, Alphabet, and Precondition

The machine is named `EQABC` and $\Sigma = \{$`a b c`$\}$.

Let `t0h` be the position of the head on tape 0. The precondition for `EQABC` is:

```
;; PRE: (LM BLANK w) AND t0h = 1 AND tapes 1-3 are empty AND
;;      t1h-t3h = 0
```

The precondition states that there is a blank before the input word and the head on tape 0 is on this blank. The other tapes are empty, and their heads are on position 0 (i.e., over each tape's first blank).

71.2 Unit Tests

The tests illustrate words that are rejected and words that are accepted. The following are a sample tests:

```
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a b b a c c) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a a) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK c c a b b) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK) 1) 'accept)
(check-equal? (sm-apply EQABC `(,LM ,BLANK a c c b a b) 1)
              'accept)
(check-equal?
 (sm-apply EQABC
           `(,LM ,BLANK c c c a b b a a c b a b b c a) 1)
 'accept)
```

Observe that all the tests satisfy the precondition. That is, for all tests, there is a blank between `LM` and the input word, and tape 0's head is on position 1 over this blank. This includes the test when the empty word is given as input (i.e., the fourth test).

71.3 Conditions and States

To determine the different conditions that are represented by a state, we must outline a design idea. How can an mttm determine if the input word has an equal number of as, bs, and cs? We shall design a machine with four tapes that operates in two phases. In the first phase, the input word is traversed. Every a is copied to tape 1, every b is copied to tape 2, and every c is copied to tape 2. In the second phase, the copied as, bs, and cs are matched. If all are matched, then the machine moves to accept. If at any point an a, b, or c cannot be matched, then the machine moves to reject.

When the machine starts, the precondition must be met. We may document, S, the starting state as follows:

```
;; S: tape 0 = (LM BLANK w) AND t0h = 1
;;    tape 1-3 = '(BLANK) AND t1h = t2h = t3h = 0
;;    starting state
```

The position of the heads on tapes 1–3 is denoted, respectively, by t1h, t2h, and t3h.

In the first phase, the machine copies all the elements on tape 0 to tapes 1–3. We use a state, C, to copy the elements one at a time. The head on tape 0 is over the next element to copy. Tape 1's touched part contains a blank, the copied as, and a blank. Its head is over the blank after the copied as at a position that corresponds to the number of as copied plus 1 (for the initial blank). This is where the next a, if any, is copied. Tapes 2 and 3 are described in a similar manner using, respectively, b and c. The role of C is documented as follows:

```
;; C: tape 0 = (LM BLANK w) AND t0h >= 2
;;    tape 1 = (BLANK a* BLANK) AND
;;      num a = num a in tape0[2..t0h-1] AND
;;      t1h = num a in tape1[2..t0h-1] + 1 AND
;;      tape1[t1h] = BLANK
;;    tape 2 = (BLANK b* BLANK) AND
;;      num b = num b in tape2[2..t0h-1] AND
;;      t2h = num b in tape2[2..t0h-1] + 1 AND
;;      tape1[t2h] = BLANK
;;    tape 3 = (BLANK c* BLANK) AND
;;      num c = num c in tape3[2..t0h-1] AND
;;      t3h = num c in tape3[2..t0h-1] + 1 AND
;;      tape1[t3h] = BLANK
```

When an element is copied from tape 0 to another tape, the machine moves to a different state depending on the element copied. A different state is needed for each of Σ's elements, because a different tape is mutated and the head on the mutated tape must be moved right to place it over the next blank where the next element, if any, may be copied. We use states D, E, and F,

respectively, for copying an a, a b, and a c. In each of these states, the element read on tape 0 has been copied to the proper auxiliary tape. For example, if on tape 0 a b is read, then on tape 2, the copied b is read, and a blank is read on the other auxiliary tapes (i.e., in state E). These states are documented as follows:

```
;; D: tape 0 = (LM BLANK w) AND
;;      t0h >= 2 AND
;;      tape0[t0h] = a
;;    tape 1 = (BLANK a* BLANK) AND
;;      num a = num a in tape1[2..t0h] AND
;;      t1h = num a in tape1[2..t0h] + 1 AND
;;      tape1[t1h] = a
;;    tape 2 = (BLANK b* BLANK) AND
;;      num b = num b in tape2[2..t0h] AND
;;      t2h = num b in tape2[2..t0h] + 1 AND
;;      tape2[t2h] = BLANK
;;    tape 3 = (BLANK c* BLANK) AND
;;      num c = num c in tape3[2..t0h] AND
;;      t3h = num c in tape3[2..t0h] + 1 AND
;;      tape3[t3h] = BLANK
;;
;; E: tape 0 = (LM BLANK w) AND
;;      t0h >= 2 AND
;;      tape0[t0h] = b
;;    tape 1 = (BLANK a* BLANK) AND
;;      num a = num a in tape1[2..t0h-1] AND
;;      t1h = num a in tape1[2..t0h-1] + 1 AND
;;      tape1[t1h] = BLANK
;;    tape 2 = (BLANK b* BLANK) AND
;;      num b = num b in tape2[2..t0h-1] AND
;;      t2h = num b in tape2[2..t0h-1] + 1 AND
;;      tape2[t2h] = b
;;    tape 3 = (BLANK c* BLANK) AND
;;      num c = num c in tape3[2..t0h-1] AND
;;      t3h = num c in tape3[2..t0h-1] + 1 AND
;;      tape3[t3h] = BLANK
;;
;; F: tape 0 = (LM BLANK w) AND
;;      t0h >= 2 AND
;;      tape0[t0h] = c
;;    tape 1 = (BLANK a* BLANK) AND
;;      num a = num a in tape1[2..t0h] AND
;;      t1h = num a in tape1[2..t0h] + 1 AND
;;      tape1[t1h] = BLANK
```

```
;;    tape 2 = (BLANK b* BLANK) AND
;;      num b = num b in tape2[2..t0h] AND
;;      t2h = num b in tape2[2..t0h] + 1 AND
;;      tape2[t2h] = BLANK
;;    tape 3 = (BLANK c* BLANK) AND
;;      num c = num c in tape3[2..t0h] AND
;;      t3h = num c in tape3[2..t0h] + 1 AND
;;      tape3[t3h] = c
```

The second phase starts when on all tapes, a blank is read. This means that all elements on tape 0 have been copied to an auxiliary tape and the matching process may start. A new state, G, is used to implement the matching process. In this state, the machine matches a's, bs, and cs one at a time, from right to left, on tapes 1–3. What is expected to be true during the matching process? Think about this carefully. Tapes 1, 2, and 3, respectively, must only contain as, bs, and cs. In addition, the number of as on tape 1 must equal the number of as on tape 0, the number of bs on tape 2 must equal the number of bs on tape 0, and the number of cs on tape 3 must equal the number of cs on tape 0. Finally, the number of matched as must equal the number of matched bs, which must equal the number of matched cs. The state may be documented as follows:

```
;; G: tape 0 = (LM BLANK w)
;;    t1 = (BLANK a*)
;;    t2 = (BLANK b*)
;;    t3 = (BLANK c*)
;;    num a in t0 = num a in t1
;;    num b in t0 = num b in t2
;;    num c in t0 = num c in t3
;;    (= |as matched| |bs matched| |cs matched|)
```

Finally, we shall have two final states: Y to accept and N to reject. Carefully reason about what needs to hold if the machine reaches Y. We must be able to conclude that the input word is in L. This means that, respectively, all as, bs, and cs on tape 0 are copied to tapes 1–3. In addition, the number of as on tape 1 must equal the number of bs on tape 2, which must equal the number of cs on tape 3. Now, carefully reason about what needs to hold if the machine reaches N. We must be able to conclude that the input word is not in L. This means that, respectively, all as, bs, and cs on tape 0 are copied to tapes 1–3. In addition, the number of as on tape 1, the number of bs on tape 2, and the number of cs on tape 3 are not equal. The states are documented as follows:

```
;; Y: num a in t0 = num a in t1
;;    num b in t0 = num b in t2
;;    num c in t0 = num c in t3
;;    num a in t1 = num b in t2 = num c in t3
;;
```

```
;; N: num a in t0 = num a in t1
;;    num b in t0 = num b in t2
;;    num c in t0 = num c in t3
;;    (not (num a in t1 = num b in t2 = num c in t3))
```

71.4 Transition Relation

From the starting state, `S`, the machine must prepare for the first phase of the computation by transitioning into `C` and moving the heads on all tapes to the right. In this manner, the head on tape 0 is over the first word element, if any, and the rest of the heads are over the blank where the next element from tape 0 may be copied. The needed transition is:

```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```

When in `C`, the machine must copy the read element on tape 0 to either tape 1, tape 2, or tape 3 and move right the heads on tape 0 and on the tape copied to. This is done in two steps. First, the element read on tape 0 is copied, and the machine moves to an intermediate state. If the element read is an `a`, it is copied to tape 1, and the machine transitions to `D`. If it is a `b`, it is copied to tape 2, and the machine transitions to `E`. If it is a `c`, it is copied to tape 3, and the machine transitions to `F`. Second, in the intermediate state, the machine transitions to `C` to continue with phase 1 of the computation and moves to the right the heads on tape 0 and on the tape copied to. The needed transitions are:

```
(list (list 'C (list 'a BLANK BLANK BLANK))
      (list 'D (list 'a 'a BLANK BLANK)))
(list (list 'D (list 'a 'a BLANK BLANK))
      (list 'C (list RIGHT RIGHT BLANK BLANK)))

 (list (list 'C (list 'b BLANK BLANK BLANK))
       (list 'E (list 'b BLANK 'b BLANK)))
 (list (list 'E (list 'b BLANK 'b BLANK))
       (list 'C (list RIGHT BLANK RIGHT BLANK)))

 (list (list 'C (list 'c BLANK BLANK BLANK))
       (list 'F (list 'c BLANK BLANK 'c)))
 (list (list 'F (list 'c BLANK BLANK 'c))
       (list 'C (list RIGHT BLANK BLANK RIGHT)))
```

If a blank is read on tape 0 when in `C`, then phase 1 of the computation has ended, and the machine transitions to `G` to start phase 2. For this, the heads on tapes 1–3 are moved left. The needed transition is:

```
(list (list 'C (list BLANK BLANK BLANK BLANK))
      (list 'G (list BLANK LEFT LEFT LEFT)))
```

In G, if blanks are read on tapes 1–3, then the word on tape 0 has an equal number of as, bs, and cs, and the machine transitions to Y to accept. If an a is read on tape 1, a b is read on tape 2, and a c is read on tape 3, then the machines remains in G and moves the heads on tapes 1–3 left to continue the matching process. For these conditions, the needed transition rules are:

```
(list (list 'G (list BLANK BLANK BLANK BLANK))
      (list 'Y (list BLANK BLANK BLANK BLANK)))

(list (list 'G (list BLANK 'a 'b 'c))
      (list 'G (list BLANK LEFT LEFT LEFT)))
```

In G, if there are too many of at least one letter, then the machine transitions to N to reject. The needed transitions are:

```
(list (list 'G (list BLANK BLANK 'b 'c))
      (list 'N (list BLANK BLANK 'b 'c)))
(list (list 'G (list BLANK 'a BLANK 'c))
      (list 'N (list BLANK 'a BLANK 'c)))
(list (list 'G (list BLANK 'a 'b BLANK))
      (list 'N (list BLANK 'a 'b BLANK)))
(list (list 'G (list BLANK BLANK BLANK 'c))
      (list 'N (list BLANK BLANK BLANK 'c)))
(list (list 'G (list BLANK BLANK 'b BLANK))
      (list 'N (list BLANK BLANK 'b BLANK)))
(list (list 'G (list BLANK 'a BLANK BLANK))
      (list 'N (list BLANK 'a BLANK BLANK))))
```

71.5 Machine Implementation and Testing

The implementation of the machine is sketched as follows:

```
;; PRE: (LM BLANK w) AND i = 1
(define EQABC
        (make-mttm
         '(S Y N C D E F G)
         '(a b c)
         'S
         '(Y N)
         (list
          (list (list 'S (list BLANK BLANK BLANK BLANK))
                (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
          ⋮
         4
         'Y)))
```

You may fill in the missing transition rules by copying them from the previous subsection. Observe that the machine is constructed with four tapes: one for the input and three auxiliary tapes for the copying and matching.

Run all the tests, and make sure they pass.

71.6 Invariant Predicates

The invariant predicates for `mttms` take as input a list of tape configurations. Each tape configuration has a natural number for the head's position on the tape and the touched part of the tape. For instance, the following is a sample list of tape configurations for a machine with four tapes:

```
(list (list 1 `(,LM ,BLANK a b c))
      (list 0 `(,BLANK a))
      (list 0 `(,BLANK b))
      (list 0 `(,BLANK)))
```

On tape 0, the head is on position 1 (i.e., the blank after the left-end marker), and on tapes 1–3, the heads are all on position 0. Next to each head position is the touched part of the tape, not including the infinite number of blanks to the right. For example, on tape 3, only the first blank has been read.

An invariant predicate must test the contents of each tape to validate that its contents satisfy the expected conditions. This means that each invariant predicate must extract the head position and the contents for each tape in the given list of configurations. We shall define these values locally using a `let*`-expression. For a machine with four tapes, an invariant predicate is outlined as follows:

```
(define (Z-INV tape-configs)
  (let* [(t0c (first tape-configs))
         (t1c (second tape-configs))
         (t2c (third tape-configs))
         (t3c (fourth tape-configs))
         (t0h (first t0c))
         (t0 (second t0c))
         (t1h (first t1c))
         (t1 (second t1c))
         (t2h (first t2c))
         (t2 (second t2c))
         (t3h (first t3c))
         (t3 (second t3c))]
    <body of let*>
```

A local variable is defined for each tape configuration (i.e., `t0c`–`t3c`), for the head position on each tape (i.e., `t0h`–`t3h`), and for the contents for each

tape (i.e., t0–t3). Instead of repeatedly writing the declaration of such local variables over and over, the presentation that follows denotes them with a vertical ellipsis as follows:

```
;; (listof tape-config) → Boolean
(define (Z-INV tape-configs)
  (let* [⋮
         <other local declarations if any>]
    <body of let*>
```

To simplify the development of the invariant predicates for EQABC, we observe that the transitions never mutate tape 0. Therefore, we shall not test that its contents are (LM BLANK w). We simply assume it is always true given that it is part of the precondition.

71.6.1 The S Invariant Predicate

The invariant predicate for S tests that the machine's initial configuration satisfies the precondition. This means that the head's position on tape 0 is 1, that the other heads are positioned at 0, a blank is read on all the tapes, and tapes 1–3 are empty. The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (S-INV tape-configs)
  (let* [⋮]
    (and (= t0h 1) (= t1h 0) (= t2h 0) (= t3h 0)
         (eq? (list-ref t0 t0h) BLANK)
         (equal? `(,BLANK) t1)
         (equal? `(,BLANK) t2)
         (equal? `(,BLANK) t3))))

(check-equal? (S-INV (list (list 1 `(,LM ,BLANK a b c))
                           (list 0 `(,BLANK a))
                           (list 0 `(,BLANK b))
                           (list 0 `(,BLANK))))
              #f)
(check-equal? (S-INV (list (list 1 `(,LM ,BLANK a b c))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))))
              #t)
```

71.6.2 The C Invariant Predicate

The invariant predicate for C tests that tape 0's head position is greater than or equal to 2, that tape 1 only contains as, that tape 2 only contains bs, and that tape 3 only contains cs. It also tests that the number of as on tape 1 equals the number of as read on tape 0, that the number of bs on tape 2 equals the number of bs read on tape 0, and that the number of cs on tape 3 equals the number of cs read on tape 0. In addition, it tests that tape 1's head position is one more than the number of as read on tape 0, that tape 2's head position is one more than the number of bs read on tape 0, and that tape 3's head position is one more than the number of cs read on tape 0. Finally, it tests that on tapes 1–3, a blank is read.

To implement this predicate, it is necessary to extract the input word's read elements on tape 0. The read elements do not include LM and the first blank on the tape. They include the next t0h-2 elements.

To test the contents of tapes 1–3, we observe that the elements written to a tape must be between two blanks. Therefore, the written elements are obtained by dropping the leading and ending blanks. All the written elements on tape 1 must be as, all the written elements on tape 2 must be bs, and all the written elements on tape 3 must be cs. Finally, each of these tapes' length must be 2 greater than the length of its written elements.

The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (C-INV tape-configs)
  (let* [⋮
         (readt0 (take (rest (rest t0)) (- t0h 2)))]
    (and (>= t0h 2)
         (eq? (list-ref t1 t1h) BLANK)
         (eq? (list-ref t2 t2h) BLANK)
         (eq? (list-ref t3 t3h) BLANK)
         (let [(written-t1 (rest (drop-right t1 1)))
               (written-t2 (rest (drop-right t2 1)))
               (written-t3 (rest (drop-right t3 1)))]
           (andmap (λ (s) (eq? s 'a)) written-t1)
           (andmap (λ (s) (eq? s 'b)) written-t2)
           (andmap (λ (s) (eq? s 'c)) written-t3)
           (= (length t1) (+ (length written-t1) 2))
           (= (length t2) (+ (length written-t2) 2))
           (= (length t3) (+ (length written-t3) 2)))
         (equal? (filter (λ (s) (eq? s 'a)) readt0)
                 (filter (λ (s) (eq? s 'a)) t1))
         (equal? (filter (λ (s) (eq? s 'b)) readt0)
                 (filter (λ (s) (eq? s 'b)) t2))
         (equal? (filter (λ (s) (eq? s 'c)) readt0)
                 (filter (λ (s) (eq? s 'c)) t3)))))
```

```
(check-equal? (C-INV (list (list 2 `(,LM ,BLANK b b b))
                           (list 0 `(,BLANK a a))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))))
              #f)
(check-equal? (C-INV (list (list 2 `(,LM ,BLANK b a c))
                           (list 1 `(,BLANK ,BLANK))
                           (list 1 `(,BLANK ,BLANK))
                           (list 1 `(,BLANK ,BLANK))))
              #t)
(check-equal?
  (C-INV (list (list 6 `(,LM ,BLANK b a b c a a b c))
               (list 2 `(,BLANK a ,BLANK))
               (list 3 `(,BLANK b b ,BLANK))
               (list 2 `(,BLANK c ,BLANK))))
  #t)
```

71.6.3 The D Invariant Predicate

The invariant predicate for D tests if tape 0's head position is greater than or equal to 2. Furthermore, it tests that on tapes 0 and 1, an a is read, and that on tapes 2 and 3, a blank is read. In addition, it tests that everything written on tapes 1–3 are, respectively, as, bs, and cs. Observe that in this state, the written elements on tape 1 include the element under tape 1's head. This means the tape 1's written part includes the $\texttt{t1h}^{\texttt{th}}$ element. The written part of tapes 2 and 3 does not include, respectively, the $\texttt{t2h}^{\texttt{th}}$ and $\texttt{t3h}^{\texttt{th}}$ elements. Finally, it also tests that the number of as on tape 1 equals the number of as read on tape 0, that the number of bs on tape 2 equals the number of bs read on tape 0, and that the number of cs on tape 3 equals the number of cs read on tape 0.

Observe that the read elements on tape 0 do not include LM and the first blank and include the next t0h-1 elements. To extract these read elements, tape 0's first two elements are dropped, and the next t0h-1 elements are taken. Further observe that the written elements on tape 1 include everything after the initial blank, and the written elements on tapes 2 and 3 are between two blanks.

The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (D-INV tape-configs)
  (let* [⋮
         (readt0 (take (rest (rest t0)) (sub1 t0h)))]
```

```
      (and (>= t0h 2)
           (eq? (list-ref t0 t0h) 'a)
           (eq? (list-ref t1 t1h) 'a)
           (eq? (list-ref t2 t2h) BLANK)
           (eq? (list-ref t3 t3h) BLANK)
           (let [(written-t1 (rest t1))
                 (written-t2 (rest (drop-right t2 1)))
                 (written-t3 (rest (drop-right t3 1)))]
             (andmap (λ (s) (eq? s 'a)) written-t1)
             (andmap (λ (s) (eq? s 'b)) written-t2)
             (andmap (λ (s) (eq? s 'c)) written-t3)
             (= (length t1) (+ (length written-t1) 1))
             (= (length t2) (+ (length written-t2) 2))
             (= (length t3) (+ (length written-t3) 2)))
           (equal? (filter (λ (s) (eq? s 'a)) readt0)
                   (filter (λ (s) (eq? s 'a)) t1))
           (equal? (filter (λ (s) (eq? s 'b)) readt0)
                   (filter (λ (s) (eq? s 'b)) t2))
           (equal? (filter (λ (s) (eq? s 'c)) readt0)
                   (filter (λ (s) (eq? s 'c)) t3)))))

(check-equal? (D-INV (list (list 2 `(,LM ,BLANK b))
                           (list 0 `(,BLANK a a))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))))
              #f)
(check-equal? (D-INV (list (list 4 `(,LM ,BLANK b a c))
                           (list 2 `(,BLANK a ,BLANK))
                           (list 2 `(,BLANK b ,BLANK))
                           (list 1 `(,BLANK ,BLANK))))
              #f)
(check-equal?
  (D-INV (list (list 2 `(,LM ,BLANK a a a b b b c c c))
               (list 1 `(,BLANK a))
               (list 1 `(,BLANK ,BLANK))
               (list 1 `(,BLANK ,BLANK))))
         #t)

(check-equal?
  (D-INV (list (list 5 `(,LM ,BLANK c a b a b a))
               (list 2 `(,BLANK a a))
               (list 2 `(,BLANK b ,BLANK))
               (list 2 `(,BLANK c ,BLANK))))
         #t)
```

71.6.4 The E Invariant Predicate

The invariant predicate for E is similar to the invariant predicate for D. Instead of using a and tape 1, b and tape 2 are used. The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (E-INV tape-configs)
  (let* [⋮
         (readt0 (take (rest (rest t0)) (- t0h 1)))]
    (and (>= t0h 2)
         (eq? (list-ref t0 t0h) 'b)
         (eq? (list-ref t1 t1h) BLANK)
         (eq? (list-ref t2 t2h) 'b)
         (eq? (list-ref t3 t3h) BLANK)
         (let [(written-t1 (rest (drop-right t1 1)))
               (written-t2 (rest t2))
               (written-t3 (rest (drop-right t3 1)))]
           (andmap (λ (s) (eq? s 'a)) written-t1)
           (andmap (λ (s) (eq? s 'b)) written-t2)
           (andmap (λ (s) (eq? s 'c)) written-t3)
           (= (length t1) (+ (length written-t1) 2))
           (= (length t2) (+ (length written-t2) 1))
           (= (length t3) (+ (length written-t3) 2)))
         (equal? (filter (λ (s) (eq? s 'a)) readt0)
                 (filter (λ (s) (eq? s 'a)) t1))
         (equal? (filter (λ (s) (eq? s 'b)) readt0)
                 (filter (λ (s) (eq? s 'b)) t2))
         (equal? (filter (λ (s) (eq? s 'c)) readt0)
                 (filter (λ (s) (eq? s 'c)) t3)))))

(check-equal? (E-INV (list (list 2 `(,LM ,BLANK b))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))))
              #f)

(check-equal? (E-INV (list (list 4 `(,LM ,BLANK a c b))
                           (list 2 `(,BLANK a ,BLANK))
                           (list 1 `(,BLANK ,BLANK))
                           (list 2 `(,BLANK c ,BLANK))))
              #f)
```

```
(check-equal?
  (E-INV (list (list 7 `(,LM ,BLANK a a a b b b c c c))
               (list 4 `(,BLANK a a a ,BLANK))
               (list 3 `(,BLANK b b b))
               (list 1 `(,BLANK ,BLANK))))
  #t)

(check-equal? (E-INV (list (list 4 `(,LM ,BLANK a c b))
                           (list 2 `(,BLANK a ,BLANK))
                           (list 1 `(,BLANK b))
                           (list 2 `(,BLANK c ,BLANK))))
              #t)

(check-equal?
  (E-INV (list (list 6 `(,LM ,BLANK c a b a b a))
               (list 3 `(,BLANK a a ,BLANK))
               (list 2 `(,BLANK b b))
               (list 2 `(,BLANK c ,BLANK))))
  #t)
```

71.6.5 The F Invariant Predicate

The invariant predicate for F is similar to the invariant predicates for D and E. Instead of using a and tape 1 or b and tape 2, c and tape 3 are used. The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (F-INV tape-configs)
  (let* [⋮
         (readt0 (take (rest (rest t0)) (- t0h 1)))]
    (and (>= t0h 2)
         (eq? (list-ref t0 t0h) 'c)
         (eq? (list-ref t1 t1h) BLANK)
         (eq? (list-ref t2 t2h) BLANK)
         (eq? (list-ref t3 t3h) 'c)
         (let [(written-t1 (rest (drop-right t1 1)))
               (written-t2 (rest (drop-right t2 1)))
               (written-t3 (rest t3))]
           (andmap (λ (s) (eq? s 'a)) written-t1)
           (andmap (λ (s) (eq? s 'b)) written-t2)
           (andmap (λ (s) (eq? s 'c)) written-t3)
           (= (length t1) (+ (length written-t1) 2))
           (= (length t2) (+ (length written-t2) 2))
           (= (length t3) (+ (length written-t3) 1)))))
```

```
          (equal? (filter (λ (s) (eq? s 'a)) readt0)
                  (filter (λ (s) (eq? s 'a)) t1))
          (equal? (filter (λ (s) (eq? s 'b)) readt0)
                  (filter (λ (s) (eq? s 'b)) t2))
          (equal? (filter (λ (s) (eq? s 'c)) readt0)
                  (filter (λ (s) (eq? s 'c)) t3)))))

(check-equal? (F-INV (list (list 2 `(,LM ,BLANK b))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))
                           (list 0 `(,BLANK))))
              #f)

(check-equal? (F-INV (list (list 2 `(,LM ,BLANK a c b))
                           (list 2 `(,BLANK a ,BLANK))
                           (list 1 `(,BLANK ,BLANK))
                           (list 1 `(,BLANK ,BLANK))))
              #f)

(check-equal?
  (F-INV (list (list 9 `(,LM ,BLANK a a a b b b c c c))
               (list 4 `(,BLANK a a a ,BLANK))
               (list 4 `(,BLANK b b b ,BLANK))
               (list 1 `(,BLANK c c ,BLANK))))
  #t)

(check-equal? (F-INV (list (list 3 `(,LM ,BLANK a c b))
                           (list 2 `(,BLANK a ,BLANK))
                           (list 1 `(,BLANK ,BLANK))
                           (list 1 `(,BLANK c ,BLANK))))
              #t)

(check-equal?
  (F-INV (list (list 7 `(,LM ,BLANK c a b a b c a))
               (list 3 `(,BLANK a a ,BLANK))
               (list 3 `(,BLANK b b ,BLANK))
               (list 2 `(,BLANK c c ,BLANK))))
  #t)
```

71.6.6 The G Invariant Predicate

The invariant predicate for G tests that the head position on each tape is less than the length of the corresponding tape length. It also tests if the same number of as, bs, and cs has been matched; if everything written on tapes 1–3 is, respectively, a, b, or c; and if all the as, bs, and cs have been copied, respectively, to tapes 1–3.

The list of as matched is empty if tape 1's length is 2 (i.e., the touched part of tape 1 only contains two blanks). Otherwise, the list of matched as is obtained by dropping the ending blank and the first t1h + 1 elements from the front of the tape. Similarly, the list of matched bs and cs is obtained using, respectively, tapes 2 and 3.

The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (G-INV tape-configs)
  (let* [⋮]
    (and (< t1h (length t1))
         (< t2h (length t2))
         (< t3h (length t3))
         (let [(as-matched
                (if (= (length t1) 2)
                    '()
                    (drop-right (drop t1 (add1 t1h)) 1)))
               (bs-matched
                (if (= (length t2) 2)
                    '()
                    (drop-right (drop t2 (add1 t2h)) 1)))
               (cs-matched
                (if (= (length t3) 2)
                    '()
                    (drop-right (drop t3 (add1 t3h)) 1)))]
           (and (= (length as-matched)
                   (length bs-matched)
                   (length cs-matched))
                (andmap (λ (s) (eq? s 'a))
                        (rest (drop-right t1 1)))
                (andmap (λ (s) (eq? s 'b))
                        (rest (drop-right t2 1)))
                (andmap (λ (s) (eq? s 'c))
                        (rest (drop-right t3 1)))
                (equal? (filter (λ (s) (eq? s 'a)) t0)
                        (filter (λ (s) (eq? s 'a)) t1))
                (equal? (filter (λ (s) (eq? s 'b)) t0)
                        (filter (λ (s) (eq? s 'b)) t2))
                (equal? (filter (λ (s) (eq? s 'c)) t0)
                        (filter (λ (s) (eq? s 'c)) t3)))))))

(check-equal? (G-INV (list (list 4 `(,LM ,BLANK b a a ,BLANK))
                           (list 0 `(,BLANK a a ,BLANK))
                           (list 0 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK ,BLANK))))
              #f)
```

```
(check-equal? (G-INV (list (list 2 `(,LM ,BLANK a c b))
                           (list 1 `(,BLANK a ,BLANK))
                           (list 0 `(,BLANK ,BLANK))
                           (list 1 `(,BLANK c ,BLANK))))
              #f)

(check-equal?
  (G-INV (list (list 11 `(,LM ,BLANK a a a b b b c c c ,BLANK))
               (list 3 `(,BLANK a a a ,BLANK))
               (list 3 `(,BLANK b b b ,BLANK))
               (list 3 `(,BLANK c c c ,BLANK))))
  #t)

(check-equal? (G-INV (list (list 5 `(,LM ,BLANK a c b ,BLANK))
                           (list 0 `(,BLANK a ,BLANK))
                           (list 0 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK c ,BLANK))))
              #t)

(check-equal?
  (G-INV (list (list 9 `(,LM ,BLANK c a b a b c a ,BLANK))
               (list 2 `(,BLANK a a a ,BLANK))
               (list 1 `(,BLANK b b ,BLANK))
               (list 1 `(,BLANK c c ,BLANK))))
  #t)
```

71.6.7 The Y Invariant Predicate

The invariant predicate for Y tests that the number of as, the number of bs, and the number of cs on tape 0 are equal. The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (Y-INV tape-configs)
  (let* [⋮
         (t0as (filter (λ (s) (eq? s 'a)) t0))
         (t0bs (filter (λ (s) (eq? s 'b)) t0))
         (t0cs (filter (λ (s) (eq? s 'c)) t0))]
    (= (length t0as) (length t0bs) (length t0cs))))
```

```
(check-equal? (Y-INV (list (list 4 `(,LM ,BLANK b a a ,BLANK))
                           (list 0 `(,BLANK a a ,BLANK))
                           (list 0 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK ,BLANK))))
              #f)

(check-equal? (Y-INV (list (list 2 `(,LM ,BLANK a a b))
                           (list 0 `(,BLANK a a ,BLANK))
                           (list 0 `(,BLANK ,BLANK))
                           (list 0 `(,BLANK c ,BLANK))))
              #f)

(check-equal?
  (Y-INV (list (list 11 `(,LM ,BLANK a a a b b b c c c ,BLANK))
         (list 0 `(,BLANK a a a ,BLANK))
         (list 0 `(,BLANK b b b ,BLANK))
         (list 0 `(,BLANK c c c ,BLANK))))
  #t)

(check-equal? (Y-INV (list (list 5 `(,LM ,BLANK a c b ,BLANK))
                           (list 0 `(,BLANK a ,BLANK))
                           (list 0 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK c ,BLANK))))
              #t)

(check-equal?
  (Y-INV (list (list 8 `(,LM ,BLANK c a b a b c ,BLANK))
         (list 0 `(,BLANK a a ,BLANK))
         (list 0 `(,BLANK b b ,BLANK))
         (list 0 `(,BLANK c c ,BLANK))))
  #t)
```

71.6.8 The N Invariant Predicate

The invariant predicate for N tests that the number of as, the number of bs, and the number of cs on tape 0 are not equal. The invariant predicate is implemented as follows:

```
;; (listof tape-config) → Boolean
(define (N-INV tape-configs)
  (let* [⋮
         (t0as (filter (λ (s) (eq? s 'a)) t0))
         (t0bs (filter (λ (s) (eq? s 'b)) t0))
         (t0cs (filter (λ (s) (eq? s 'c)) t0))]
    (not (= (length t0as) (length t0bs) (length t0cs)))))
```

```
(check-equal? (N-INV (list (list 5 `(,LM ,BLANK b c a ,BLANK))
                           (list 0 `(,BLANK a ,BLANK))
                           (list 0 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK c ,BLANK))))
              #f)

(check-equal? (N-INV (list (list 2 `(,LM ,BLANK ,BLANK))
                           (list 0 `(,BLANK ,BLANK))
                           (list 0 `(,BLANK ,BLANK))
                           (list 0 `(,BLANK ,BLANK))))
              #f)

(check-equal? (N-INV (list (list 5 `(,LM ,BLANK b a a ,BLANK))
                           (list 2 `(,BLANK a a ,BLANK))
                           (list 1 `(,BLANK b ,BLANK))
                           (list 0 `(,BLANK ,BLANK))))
              #t)

(check-equal?
  (N-INV (list (list 7 `(,LM ,BLANK b a a c b ,BLANK))
               (list 1 `(,BLANK a a ,BLANK))
               (list 1 `(,BLANK b b ,BLANK))
               (list 0 `(,BLANK c ,BLANK))))
  #t)
```

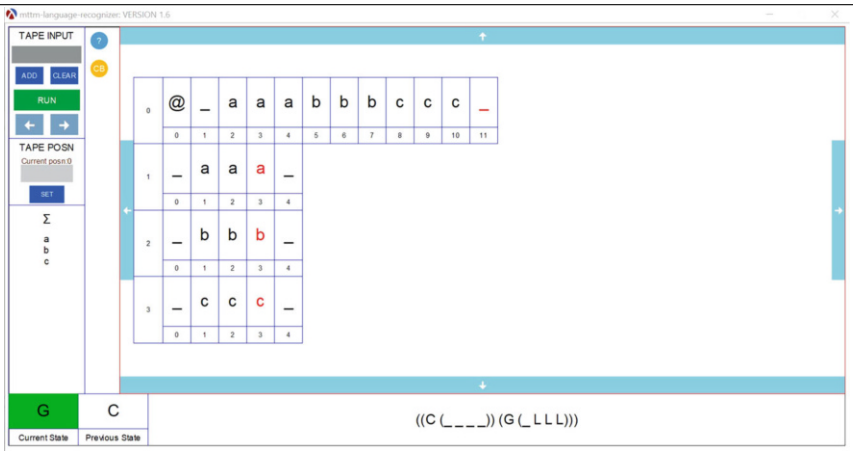
71.7 Visualizing mttms

The visualization tool displays mttms differently from previous state machines. Figure 94 displays a snapshot of running EQABC with invariant predicates:

```
(sm-visualize EQABC
              (list 'S S-INV)
              (list 'Y Y-INV)
              (list 'N N-INV)
              (list 'C C-INV)
              (list 'D D-INV)
              (list 'E E-INV)
              (list 'F F-INV)
              (list 'G G-INV))
```

On the bottom, the current state, the previous state, and the last transition rule are displayed. The current state's enclosing box is colored green if the invariant for the current state holds and is colored red if the invariant for

Fig. 94 An `mttm` visualization with invariants



the current state does not hold. In Fig. 94, `G`'s invariant holds. If an invariant predicate is not provided for a state, then the current state's enclosing box is not colored.

The left column allows the programmer to add elements to the input tape. Recall that to add a blank, you type `BLANK`. In the left column, the programmer may also set the initial position of the head on tape 0. Finally, the left column also displays the machine's alphabet.

The center displays an image containing the configuration of each tape. For each tape, the touched part of the tape is displayed along with the index of each element. The element under the head is displayed in red. In Fig. 94, for example, tape 2 contains a blank, 3 `b`s, and a blank. Tape 2's head is over the `b` in position 3.

As for other state machines, run the visualization tool to validate invariant predicates and the transitions before attempting to prove correctness. You ought to use the visualization tool to validate `EQABC`'s invariant predicates. Once you are fairly certain that the invariant predicates always hold, proceed to developing a proof that establishes that they do and a proof that the machine accomplishes its goal (i.e., decide a language or compute a function).

71.8 Proving `L(EQABC) = L`

After using the visualization tool to validate `EQABC`'s invariant predicates, we are cautiously optimistic that they always hold. We may now proceed with the final step of the design recipe for state machines and develop a proof that `L(EQABC) = L`.

71.8.1 Proving Invariants Hold

The first step is to prove that EQABC's invariant predicates always hold. We do so by induction on, n, the number of transition rules used during a computation. We denote tapes 0–3 as t0–t3, the head position on each tape as t0h–t3h, and the input word as w.

Theorem 1 *The state invariants hold when EQABC is applied to w.*

Proof

Base case: n = 0
When EQABC starts, it can only reach S, given that there are no empty transitions out of S. By precondition, tape 0 contains the left-end marker, a blank, and w, t0h is 1, tapes 1–3 are empty, and t1h–t3h all equal 0. Therefore, S-INV holds.

Inductive Step:
Assume: State invariants hold for a computation of length n = k
Show: State invariants hold for a computation of length n = k + 1

((S (BLANK BLANK BLANK BLANK)) (C (R R R R)): By inductive hypothesis, S-INV holds. This means that tape 0 contains the left-end marker, a blank, and w, t0h is 1, tapes 1–3 are empty, and t1h–t3h all equal 0. Using this rule means that a blank is read on all tapes, and all heads are moved right. After using the rule, t0h is greater than or equal to 2. Given that tapes 1–3 are empty, a blank is read in the new position of each head, their written part is empty, each tape's length is the length of each tape's written part plus 2, and all read as, bs, and cs on tape 0 (i.e., 0 of each) are copied, respectively, to tapes 1–3. Therefore, C-INV holds.

((C (a BLANK BLANK BLANK)) (D (a a BLANK BLANK))): By inductive hypothesis, C-INV holds. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tapes written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule means that the a read on tape 0 is copied to tape 1. After using this rule, we have that tapes 0's head position is greater than or equal to 2; that an a is read on tape 0 and tape 1; that a blank is read on tapes 2 and 3; that the written part of tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 1 is its written part length plus 1; that the length of tapes 2 and 3 are the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Therefore, D-INV holds.

((D (a a BLANK BLANK)) (C (R R BLANK BLANK))): By inductive hypothesis, D-INV holds. This means that tapes 0's head position is greater

than or equal to 2; that an a is read on tape 0 and tape 1; that a blank is read on tapes 2 and 3; that the written part of tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 1 is its written part length plus 1; that the length of tapes 2 and 3 is the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule means that the heads on tape 0 and 1 are moved right. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tape's written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Thus, C-INV holds.

((C (b BLANK BLANK BLANK)) (E (b BLANK b BLANK))): By inductive hypothesis, C-INV holds. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tapes written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule means that the b read on tape 0 is copied to tape 2. After using the rule, we have that tapes 0's head position is greater than or equal to 2; that an b is read on tape 0 and tape 2; that a blank is read on tapes 1 and 3; that the written part of tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 2 is its written part length plus 1; that the length of tapes 1 and 3 are the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Therefore, E-INV holds.

((E (b BLANK b BLANK)) (C (R BLANK R BLANK))): By inductive hypothesis, E-INV holds. This means that tapes 0's head position is greater than or equal to 2; that an b is read on tape 0 and tape 2; that a blank is read on tapes 1 and 3; that the written part of tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 2 is its written part length plus 1; that the length of tapes 1 and 3 are the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule means that the heads on tape 0 and 2 are moved right. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tape's written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Thus, C-INV holds.

((C (c BLANK BLANK BLANK)) (F (c BLANK BLANK c))): By inductive hypothesis, C-INV holds. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tapes written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule means that the c read on tape 0 is copied to tape 3. After using the rule, we have that tapes 0's head position is greater than or equal to 2; that an c is read on tape 0 and tape 3; that a blank is read on tapes 1 and 2; that the written part of

tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 3 is its written part length plus 1; that the length of tapes 1 and 2 are the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Therefore, F-INV holds.

((F (c BLANK BLANK c)) (C (R BLANK BLANK R))): By inductive hypothesis, F-INV holds. This means tapes 0's head position is greater than or equal to 2; that an c is read on tape 0 and tape 3; that a blank is read on tapes 1 and 2; that the written part of tapes 1–3 only contain, respectively, as, bs, and cs; that length of tape 3 is its written part length plus 1; that the length of tapes 1 and 2 are the length of its written part plus 2; and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Using this rule moves the heads on tape 0 and on tape 3 to the right. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tape's written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Thus, C-INV holds.

((C (BLANK BLANK BLANK BLANK)) (G (BLANK L L L))): By inductive hypothesis, C-INV holds. This means that t0h is greater than or equal to 2, that a blank is read on tapes 1–3, that the length of tapes 1–3 is the length of each tapes written part plus 2, and that all read as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. This rule moves the heads on tapes 1–3 left. This means that the head's position on tapes 1–3 is less than the corresponding tape's length; that there is an equal number of matched as, bs, and cs on tapes 1–3 (observe that the matched elements are to the right of the respective head positions); that tapes 1–3, respectively, only contain as, bs, and cs; and that all as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Therefore, G-INV holds.

((G (BLANK BLANK BLANK BLANK)) (Y (BLANK BLANK BLANK BLANK))): By inductive hypothesis, G-INV holds. This means that the head's position on tapes 1–3 is less than the corresponding tape's length; that there is an equal number of matched as, bs, and cs on tapes 1-3; that tapes 1–3, respectively, only contain as, bs, and cs; and that all as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. This rule reads a blank on all tapes and moves to Y to accept. A blank read on tapes 1–3 means that all as, bs, and cs on these tapes are matched. Given that tapes 1–3, respectively, contain all the as, bs, and cs in w on tape 0, we have that w has an equal number of as, bs, and cs. Therefore, Y-INV holds.

((G (BLANK a b c)) (G (BLANK L L L))): By inductive hypothesis, G-INV holds. This means that the head's position on tapes 1–3 is less than the corresponding tape's length; that there is an equal number of matched as, bs, and cs on tapes 1–3; that tapes 1–3, respectively, only contain as, bs, and cs;

and that all as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. This rule moves the heads on tapes 1–3 to the left, because one more a, b, and c are matched. Thus, G-INV holds.

All other transitions out of G: By inductive hypothesis, G-INV holds. This means that the head's position on tapes 1–3 is less than the corresponding tape's length; that there is an equal number of matched as, bs, and cs on tapes 1–3; that tapes 1–3, respectively, only contain as, bs, and cs; and that all as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3. Each of these rules transition to N, because there is an unmatched a on tape 1, an unmatched b on tape 2, or an unmatched c on tape 3. Given that all as, bs, and cs on tape 0 are copied, respectively, to tapes 1–3, we have that w does not have an equal number of as, bs, and cs. Thus, N-INV holds. □

71.8.2 L(EQABC) = L

The second step is to prove L(EQABC) = L knowing that the invariant predicates hold. As before, the proof is divided in two lemmas.

Lemma 1 *w∈L ⇔ w∈L(EQABC)*

Proof
(⇒) Assume w∈L.

w∈L means that w has an equal number of as, bs, and cs. Given that invariants always hold, there is a computation that copies all as, bs, and cs, respectively, to tapes 1–3, that matches all the elements on tapes 1–3, and that moves to Y. Thus, w∈L(EQABC).

(⇐) Assume w∈L(EQABC).

w∈L(EQABC) means that there is a computation that consumes w and halts in Y. Given that invariants always hold, w has an equal number of as, bs, and cs. Therefore, w∈L. □

Lemma 2 *w∉L ⇔ w∉L(EQABC)*

Proof
(⇒) Assume w∉L.

w∉L means that w does not have an equal number of as, bs, and cs. Given that invariants always hold, there is a computation that copies all as, bs, and cs, respectively, to tapes 1–3, that fails to match all the elements on tapes 1–3, and that moves to N. Thus, w∉L(EQABC).

(⇐) Assume w∉L(EQABC).

w∉L(EQABC) means that there is a computation that consumes w and halts in N. Given that invariants always hold, w does not have an equal number of as, bs, and cs. Therefore, w∉L. □

Theorem 2 *L(EQABC) = L*

Proof
Lemmas 1 and 2 establish the theorem. □

1 Design and implement an mttm to decide L = {w | w∈$a^n b^n c^n$}.

2 Design and implement an mttm to decide L = {w | w has an equal number of as, bs, cs, and ds.}.

3 Let Σ = {a b}. Design and implement an mttm to decide L = {www | w∈Σ^*}.

4 Design and implement an mttm to multiply two unary numbers.

5 Let Σ = {a b}. Design and implement an mttm to compute f(w) = ww.

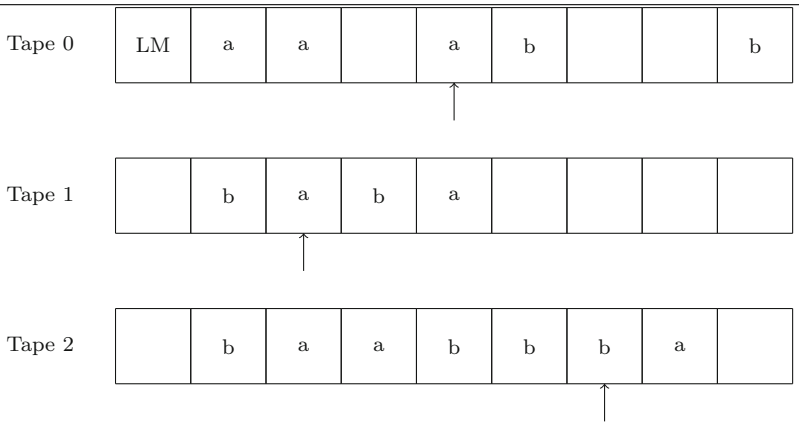
72 tm and mttm Equivalence

Multitape Turing machines are capable of performing complex computations and in many cases make the design of a solution easier. Are they more powerful than standard Turing machines? That is, is there any language that can be decided or any function computed by an mttm that cannot be decided nor computed by a tm? Intuitively, you may want to believe that with multiple tapes, there are tasks that can only be performed by an mttm. Perhaps surprisingly, the answer to our questions is no. Multiple tapes do not provide us with the ability solve problems that cannot be solved by a standard Turing machine.

This means that a standard Turing machine can simulate any multitape Turing machine. Such simulations are useful tools in studying the computational power of state machines. Typically, a simulating machine does in multiple steps what is done by the simulated machine in one step.

We shall sketch how to build a standard Turing machine from a multitape Turing machine. The word *sketch* is used because we shall not implement the constructor. There are several reasons for this. First, the details to build the simulating tm get messy very quickly. These details provide little insight into understanding that mttms may be simulated by a tm. Second, there are

Fig. 95 Tape configurations for a 3-tape `mttm`



several design choices that may be made. Making such choices are necessary to implement a constructor, but are not necessary to understand that `mttm`s may be simulated by a standard `tm`. Finally, there are limitations `FSM` imposes on us. For example, simulating an `mttm` usually requires a rich alphabet that is much larger than the set of lowercase letters in the Roman alphabet and the digits. Therefore, we are unable to encode the needed alphabet in `FSM`.

72.1 Design Idea

The tape of a `tm` that simulates an `mttm` must represent all the information contained in the `mttm`'s tape configurations. This information includes each tape's contents and each tape's head position. For instance, consider the tapes for a 3-tape `mttm` displayed in Fig. 95. The arrows indicate each tape's head position, and the empty tape locations represent a blank.

How can this information be represented on a single tape? We shall draw inspiration from hard disk technology, and imagine that a standard `tm`'s tape is divided into tracks. Specifically, if a k-tape `mttm` is simulated, then there are 2k tracks. The even tracks, numbered 2j, represent tape j's content in the `mttm`. The odd tracks, numbered 2j+1, capture the head's position on tape j. An odd track, 2i+1, contains all zeroes and a single one indicating the head's position on track i. The standard `tm` tape contains the left-end marker, the tracks for all tapes up to and including the largest position touched by any simulated head, and an infinite number of blanks to the right. Figure 96 displays the standard `tm` tape for the `mttm`–tapes displayed in Fig. 95.

The division into tracks, of course, is an abstraction that must be implemented. We can observe that alphabet for the simulating `tm` must contain the `mttm`'s alphabet. In this manner, the simulating `tm` can receive the same

Fig. 96 3-tape mttm tape representation in a tm

LM	LM	a	a		a	b			b	
	0	0	0	0	1	0	0	0	0	
		b	a	b	a					
	0	0	1	0	0	0	0	0	0	
		b	a	a	b	b	b	a		
	0	0	0	0	0	0	1	0	0	

input as the mttm. In addition, we can observe that each position in the simulating tm's tape is either LM, BLANK, or a symbol representing an element in $(\Sigma \times \{0\ 1\})^k$, where Σ is the mttm's alphabet. In summary, the simulating tm's alphabet must have a symbol for each possible column in the track representation of the multiple tapes. By scanning the portion of the tape that contains these symbols, it can be determined what symbols are read on each tape and use a state to remember them. For example, there are nine different columns formed by the tracks in Fig. 96. This means that nine different symbols, say 'p–'x, are needed to represent them. Mapping the columns from left to right to these symbols yields the following representation for a standard Turing machine's tape:

LM	p	q	r	s	t	u	v	w	x	

Assume P is a state representing that none of the read elements by the simulated mttm are known. In P, the tape is scanned from the left-end marker to the first blank. Upon reading r, the tm changes to state R, which represents that an a is read on the mttm's tape 1. Upon reading t, the tm changes to state T, which represents that an a is read on the mttm's tape 1 and an a on tape 0. Finally, upon reading v, the tm changes to state U, which represents that an a is read on the mttm's tape 1, an a on tape 0, and a b on tape 2. In summary, all the elements read by the multitape Turing machine can be determined by scanning the tape and can be remembered using states.

In a similar fashion, the mttm's state may also be coded into the simulating tm's state. Knowing both the mttm's state and the elements read by the mttm means the simulating tm may identify which rules may be used by the mttm.

As with all tms, if more than one mttm rule may be used, then a rule is nondeterministically chosen.

72.2 Proof Sketch

Theorem 3 *Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.*

Proof (sketch)
Assume the precondition for M is:

```
;; PRE: tape 0 = (LM BLANK w) and t0h = 1
```

We partially outline how the simulating tm operates. There are three primary phases M' operates in:

1. Make the M''s tape represent M's initial configuration.
   a. Shift w to the right one space
   b. To represent the beginning of the k tapes, move to the first blank, and write Σ''s symbol for:

      ```
      LM
      0
      BLANK
      1
      ⋮
      BLANK
      1
      ```

      This symbol captures that in the simulated mttm, the head on tape 0 is not on the left-end marker and that all the heads on the other tapes are on the first tape position reading a blank.
   c. Move the head to the write to capture tape 0's head position by writing Σ''s symbol for:

      ```
      BLANK
      1
      BLANK
      0
      ⋮
      BLANK
      0
      ```

d. Proceed to the right until a blank is read. For each e$\in\Sigma$ read, write Σ's symbol for:

```
e
0
BLANK
0
⋮
BLANK
0
```

2. Simulate M. For each transition that would be made by M, M' starts on the first blank that is not encoded into tracks and performs the following operations:
   a. Move left and determine what would be read by each of M's head as described in the design idea, and return the head to the first blank that is not encoded into tracks.
   b. Move left to update the tracks as M would update its tapes and heads. □
3. If M would halt, convert the contents of the tape (i.e., M''s tape) to single track. This conversion ignores all tracks except tracks 0 and 1. The content of track 0 is written to the tape, and the head is placed at the position indicated by track 1. Finally, M' moves to the state M would halt in and halt itself.

Many design choices are not described for phase 2, but it ought to be clear that M' simulates M. M' performs many transitions to simulate one transition performed by M, but this is not a concern for us in this context. The important point is that any language decided, any language semi-decided, and any function computed by an `mttm` can also be, respectively, decided, semi-decided, and computed by a standard `tm`.

73 Turing Machines and Pushdown Automata: Programming Project

Section 63 puts forth the idea that `tm`s can do everything `pda`s do. That is, given a `pda` that semi-decides or decides a language, there is a `tm` that does the same. Building such a `tm` is a long error-prone exercise in design. Instead of designing a `tm`, you shall use the lessons in this chapter. Specifically, you shall design, implement, validate, and verify an `mttm` constructor that takes as input a `pda` and returns an `mttm` that simulates the given `pda`.

Intuitively, a 2-tape `mttm` can simulate a `pda`. Tape 0 contains the input word. If the input word is empty, then tape 0 is `` `(,BLANK) ``. The `mttm` simulates the `pda` reading from the input tape by moving the head on tape 0 to the right. Tape 1 is used to simulate the `pda`'s stack, such that the stack elements written from left to right are the bottommost to the topmost. If the stack is empty, then tape 1 is `` `(,BLANK) ``. A `pda` push is simulated by writing the pushed elements to tape 1. A `pda` pop is simulated by blanking out the popped elements on tape 1 as tape 1's head is moved left. To simplify the implementation of pushing and popping in the `mttm`, it may be useful to convert the given `pda` to a simple `pda`.

Developing the required `mttm` constructor is left as a programming project. Use your answers to the following problems to guide your design, implementation, validation, and verification.

6 How does the `mttm` start a computation?

7 The given `pda` may have several final states. Reaching any of these final states does not mean the computation must halt. For `mttm`s, on the other hand, reaching a final state means the machine must halt. Furthermore, after a computation reaches a halting state, the pda accepts only if the word is consumed and stack is empty and, otherwise, rejects. Carefully outline how the `mttm` that simulates the given `pda` accepts and rejects.

8 What are the states for the `mttm`? What is the relationship with the `pda` states?

9 What is the `mttm`'s input alphabet?

10 How is the constructed `mttm` tested?

11 Implement the constructor to build an `mttm` from a `pda`.

12 Prove that the constructor is correct.

74 Other Turing Machine Extensions

74.1 Multiple Heads

Does a Turing machine with one tape and multiple heads operating on the tape provide us with more computational power? Each head operates independently of the others. If a head moves right or left, it does not interfere with other heads. A synchronization mechanism must be implemented when

two or more heads try to write to the same tape position. We shall not concern ourselves with the details of synchronization and assume that there is a synchronization policy.

How can a standard `tm` simulate a Turing machine with one tape and multiple heads? A representation strategy similar to the one used to simulate multiple tapes may be used. The standard Turing machine operates on an encoded multiple track tape. The first track represents the input tape, and the rest of the tracks record each head's position. To simulate one step of the multiple heads machine, the standard `tm` scans the tape twice. The first to determine the elements read by each of the multiple heads. The second to simulate the action of each head (i.e., mutate the first track or move the head). Although implementation details are not presented, it becomes clear that a standard `tm` can simulate a Turing machine with multiple heads.

13 Describe a Turing machine with multiple heads that makes a copy of its input word. You may assume the input word contains no blanks in it.

74.2 Two-Way Infinite Input Tape

Is a Turing machine with a tape that is infinite both to the right and the left more powerful than a standard Turing machine? In such a machine, at the beginning, only the input is written on the tape, and the rest of the positions are blank. Clearly, there is no left-end marker in such a machine.

A two-way infinite input tape Turing machine may be simulated by a 2-tape `mttm`. Initially, the first tape contains the input, and the second tape is blank. During a computation, if the head of the two-way infinite input tape Turing machine is on the first element of the input or to right of it, then the standard `tm` operates on tape 0. Otherwise, it operates on tape 1, where the elements are written in reversed order. When the two-way infinite input tape Turing machine is operating anywhere before the input, right and left moves are simulated, respectively, by left and right moves on tape 1 by the 2-tape `mttm`.

Chapter 18
Context-Sensitive Grammars

We have seen that Turing machines are automata that are capable of deciding languages that are not context-free. That is, Turing machines can be language recognizers. In previous chapters, we have also seen that less powerful automata, `dfa`s and `pda`s, have language generator counterparts called, respectively, regular grammars/expressions and context-free grammars. In fact, we have seen that languages may be characterized by an automaton or by a grammar. This chapter does the same for Turing machine language recognizers.

We shall now explore *context-sensitive grammars* (`csg`s). These grammars are also known as unrestricted grammars and as rewriting systems. Furthermore, we shall discover the equivalence between `csg`s and Turing machine language recognizers. Like previously studied grammars, a `csg` has a set of nonterminal symbols, an alphabet, a set of production rules, and a starting nonterminal. To generate a word, the left-hand side of a rule appearing in a partially generated word is substituted by the right-hand side of the rule until no more substitutions can be made. Unlike previous grammars, however, the left-hand side of a rule may not consist of a single nonterminal symbol. In a `csg`, the left-hand side of a rule may consist of any number of terminal and nonterminal symbols as long as there is at least one nonterminal symbol. In a single derivation step, all the symbols on the left-hand side of a rule are substituted with all the symbols in the right-hand side of the rule. Observe that this means that the length of a partially generated word may arbitrarily decrease in a single step and that, as with other grammars, a generated string may only contain terminal symbols.

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_18

75 Formal Definition

The name context-sensitive arises from the fact that the substitution of a nonterminal may depend on the symbols around it. What is before and after a nonterminal defines how it may be substituted. Context-sensitive grammars are a type in `FSM`. We formally define a `csg` grammar as follows:

```
A context-free grammar is an instance of (make-csg N Σ R S)
```

`N` is the set of capital letters in the Roman alphabet representing the nonterminal symbols. Σ is the set of lowercase symbols in the Roman alphabet called the alphabet (or terminal symbols). `S` is the starting nonterminal symbol, and it must be a member of `N`. `R` is a finite set of production rules. Each production rule is of the form:

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$$

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow \texttt{EMP}$$

That is, there is at least one nonterminal and an arbitrary number of terminal symbols on the left-hand side of a production rule, and there is a symbol that is either `EMP` or consists of one or more nonterminals and terminals on the right-hand side of a production rule. Observe that the set of regular grammars and the set of context-free grammars are both proper subsets of the set of context-sensitive grammars. This follows by observing that such grammars place restrictions on the structure of production rules that do not violate the definition of a context-sensitive grammar production rule.

A derivation consists of one or more derivation steps. A derivation step is the application of a production rule and is denoted by $\rightarrow_G$ (or simply $\rightarrow$ if `G` is clear from the context). Zero or more derivation steps are denoted by $\rightarrow_G^*$ (or simply $\rightarrow^*$ if `G` is clear from the context). `L(G)` denotes the language generated by `G`: $\{w \mid w \in \Sigma^* \wedge S \rightarrow_G^* w\}$. Finally, a language, `L`, is context-sensitive if `L = L(G)` for some context-sensitive grammar `G`.

To make `csg`s more tangible, consider the grammar displayed in Fig. 97. This grammar generates words in the context-free language $L = a^n b^n$. It has three nonterminals, `S`, `A`, and `B`, of which `S` is the starting nonterminal. The alphabet is {`a b`}. There are four production rules. The first production states that `S` generates `AAaAAB`: two nonterminal symbols followed by a terminal symbol followed by three nonterminal symbols. The last production rule states that a `B` generates an `A`. These two production rules are similar to any production rules in a context-free grammar (i.e., their left-hand sides consist of a single nonterminal). The remaining two production rules are more interesting. They define what `AAaAAB` generates: either `EMP` or `aSb`. These production rules differ from any production in a context-free grammar given that their left-hand sides have more than a single nonterminal symbol. As you can see, a nonterminal like `A` cannot be substituted on its own. Substitution of `A`s may only be done in the context of `AAaAAA`. Outside of that context, an

Fig. 97 A context-sensitive grammar for $a^n b^n$

```
;; L = a^nb^n
(define anbn (make-csg '(S A B)
                       '(a b)
                       `((S ,ARROW AAaAAB)
                         (AAaAAA ,ARROW aSb)
                         (AAaAAA ,ARROW ,EMP)
                         (B ,ARROW A))
                       'S))

(check-equal? (grammar-derive anbn '()) '(S -> AAaAAB -> AAaAAA -> ϵ))
(check-equal? (grammar-derive anbn '(a a a b b b))
              '(S
                -> AAaAAB
                -> AAaAAA
                -> aSb
                -> aAAaAABb
                -> aAAaAAAb
                -> aaSbb
                -> aaAAaAABbb
                -> aaAAaAAAbb
                -> aaaSbbb
                -> aaaAAaAABbbb
                -> aaaAAaAAAbbb
                -> aaabbb))
```

A cannot be substituted. Further, observe that, in the derivation displayed in the second test, the partially generated word's length may shrink by more than 1 after one step.

76 A csg for L = $a^n b^n c^n$

Clearly, a csg is not needed to generate $a^n b^n$. Such a grammar, however, offers a gentle first exposure to csgs. Let us now tackle the design and implementation of a grammar for a language that is not context-free: L = $a^n b^n c^n$. If we are successful, then we will have established that computationally csgs are more powerful than cfgs.

We shall design the grammar following the steps of the design recipe for grammars displayed in Fig. 50. When designing csgs, however, the syntactic category represented by a nonterminal may be less clear-cut than their counterparts in cfgs and rgs. For instance, a nonterminal may not represent a syntactic category on its own, given that it may only generate something in the proper context. In this regard, it is sometimes useful to think of a nonterminal as a promise to generate a desired subword in the proper context. Alternatively, you may think of nonterminals as partially defining a syntactic category. In addition, we must also take extra care in writing tests. Testing

words in the given language is easier than words not in the language. Be mindful that finding a derivation is computationally intensive especially if the grammar uses nondeterminism. Therefore, finding a derivation may take very long, making some tests unpractical. Further complicating the landscape, attempting to find derivations for a word not in the language may take forever. Again, this is especially true when the grammar uses nondeterminism. The search may continuously generate longer partially generated words in the hope of finding a derivation. This process is futile if the word is not in the language of the grammar and ends in an infinite search. Unlike for `rg`s and `cfg`s, there is no algorithm to distinguish when a search has become futile. This has rather important consequences for us as programmers. Think about this carefully. Is a search using `grammar-derive` taking a long time to find a derivation, or is it hopelessly searching for a derivation that does not exist? There is no general algorithm to decide if there is a bug in a `csg` and, therefore, the search for a derivation is futile or if there is no bug in the grammar and the search is simply computationally intensive forcing us to be patient. In such a scenario, careful design is of paramount importance.

76.1 Design Idea

The grammar may generate an arbitrary but equal number of nonterminals, say `A`, `B`, and `C`, which represent promises to generate `a`s, `b`s, and `c`s in the proper context. This generation ends with a nonterminal, say `G`, which represents a promise to generate `c`s in the proper context. The proper context, of course, is that all the `A`s must be before the all the `B`s, which must be before all the `C`s with the `G` at the end. Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the partially generated word to be in the right order. Finally, the grammar traverses the partially generated word, from right to left, to generate the `c`s. If there is `CG` in the partially generated word, then the `G` may be moved left, and a `c` may also be generated. That is, `CG` generates `Gc`. After the last `c` is generated, the grammar generates a nonterminal, say `H`, which represents a promise to generate the `b`s in the proper context and continues the traversal to generate all the `b`s. The promise is similar to the process to generate the `c`s. When the last `b` is generated, the grammar generates a nonterminal, say `I`, which represents a promise to generate the `a`s in the proper context and continues the traversal to generate all the `a`s. When the last `a` is generated, `I` generates `EMP`, and the derivation is done.

Commonly, it is useful to outline the steps to generate a concrete word. Let us consider the derivation for `aaabbbccc`. The grammar first (nondeterministically) generates `ABC` three times ending with `G`. The partially generated word is:

```
ABCABCABCG
```

Next, the grammar rearranges the As, Bs, and Cs to be in the right order. After this step, the generated word is:

```
AAABBBCCCG
```

The next step is to generate the cs and move the G left after each c is generated. The partially generated words after each c is generated are:

```
AAABBBCCGc
AAABBBCGcc
AAABBBGccc
```

Given that all the cs are generated, the grammar nondeterministically generates an H from G to generate the bs. The partially generated word becomes:

```
AAABBBHccc
```

The next step is to generate the bs and move the H left after each b is generated. The partially generated words after each b is generated are:

```
AAABBHbccc
AAABHbbccc
AAAHbbbccc
```

Given that all the bs are generated, the grammar nondeterministically generates an I from H to generate the as. The partially generated word becomes:

```
AAAIbbbccc
```

The next step is to generate the as and move the I left after each a is generated. The partially generated words after each a is generated are:

```
AAIabbbccc
AIaabbbccc
Iaaabbbccc
```

Given that all the as are generated, the grammar nondeterministically generates an EMP from I. The partially generated word becomes:

```
aaabbbccc
```

Observe that the partially generated word only contains terminal symbols. Therefore, no more substitutions are possible, and the derivation is completed. You can observe that, indeed, the desired word is generated.

76.2 Name, Alphabet, and Syntactic Categories

A descriptive name for the grammar is anbncn. The alphabet is {a b c}.

The nonterminals are syntactic categories that promise to generate a subword in the proper context. A and I together may generate an a in the context

AI. B and H together may generate a b in the context BH. C and G together may generate an a in the context CG. We document the syntactic categories as follows:

```
;; Syntactic Categories
;; S: generates words in a^nb^nc^n
;; A,I: A promise to generate an a in the context AI
;; B,H: A promise to generate an b in the context BH
;; C,G: A promise to generate an c in the context CG
```

76.3 Production Rules

S needs to generate an arbitrary number of ABC and needs to generate G. The needed production rules are:[14]

```
(S ,ARROW ABCS)
(S ,ARROW G)
```

The first rule generates one or more ABC. The second rule generates zero ABC and a G.

To rearrange the As, Bs, and Cs in the right order, production rules are needed that generate from any two of these nonterminals out of order the same nonterminals in the right order. The needed production rules are:

```
(BA ,ARROW AB)
(CA ,ARROW AC)
(CB ,ARROW BC)
```

C and G are used to generate cs and move G right to left. After all the cs are generated nondeterministically, an H must be generated. The needed production rules are:

```
(CG ,ARROW Gc)
(G  ,ARROW H)
```

B and H are used to generate bs and move H right to left. After all the bs are generated nondeterministically, an I must be generated. The needed production rules are:

```
(BH ,ARROW Hb)
(H ,ARROW I)
```

A and I are used to generate as and move I right to left. After all the as are generated nondeterministically, EMP must be generated from I. The needed production rules are:

```
(AI ,ARROW Ia)
(I ,ARROW ,EMP)
```

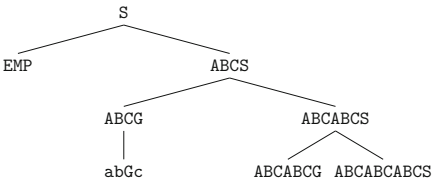
[14] We assume the production rules are inside a quasiquoted list.

76.4 Tests

Tests need to be written for words that are in the language. As noted before, finding a derivation may be computationally intensive, and therefore, some trial and error may be needed to make sure tests run in a reasonable amount of time. Usually, a good heuristic is to test relatively short words that are in the language. The following are sample tests using short words in L:

```
(check-equal? (grammar-derive anbncn-csg '())
              '(S -> G -> H -> I -> ϵ))
(check-equal? (grammar-derive anbncn-csg '(a a b b c c))
              '(S
                -> ABCS
                -> ABCABCS
                -> ABACBCS
                -> ABABCCS
                -> AABBCCS
                -> AABBCCG
                -> AABBCGc
                -> AABBGcc
                -> AABBHcc
                -> AABHbcc
                -> AAHbbcc
                -> AAIbbcc
                -> AIabbcc
                -> Iaabbcc
                -> aabbcc))
```

Writing tests using words that are not in L is ill-advised in this case. The search space for finding a derivation may be visualized as a tree. The root of the tree is the partially derived word using 0 production rules. At each level, n, of the tree, all the partially derived words using n transitions are found. Consider trying to find a derivation for aa – a word that is not in the language. Let us examine the top of the tree that represents the search space:



Observe that when a left branch is followed, the derivation represented will eventually fail. That is fine, because the search is finite. The only derivation that never fails is the one represented by the path that always takes the right branch in the tree. This is a problem, because the partially generated word is

Fig. 98 A `csg` for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

```
;; Syntactic Categories
;; S: generated words in a^nb^nc^n
;; A,I: A promise to generate an a in the context AI
;; B,H: A promise to generate an b in the context BH
;; C,G: A promise to generate an c in the context CG

(define anbncn-csg
  (make-csg '(S A B C G H I)
            '(a b c)
            `((S ,ARROW ABCS)
              (S ,ARROW G)
              (BA ,ARROW AB)
              (CA ,ARROW AC)
              (CB ,ARROW BC)
              (CG ,ARROW Gc)
              (G  ,ARROW H)
              (BH ,ARROW Hb)
              (H ,ARROW I)
              (AI ,ARROW Ia)
              (I ,ARROW ,EMP))
            'S))

(check-equal? (grammar-derive anbncn-csg '())
              '(S -> G -> H -> I -> ϵ))
(check-equal? (grammar-derive anbncn-csg '(a a b b c c))
              '(S
                -> ABCS
                -> ABCABCS
                -> ABACBCS
                -> ABABCCS
                -> AABBCCS
                -> AABBCCG
                -> AABBCGc
                -> AABBGcc
                -> AABBHcc
                -> AABHbcc
                -> AAHbbcc
                -> AAIbbcc
                -> AIabbcc
                -> Iaabbcc
                -> aabbcc))
```

always made longer in the hope of finding a derivation and inevitably leads to an infinite search. Therefore, we are unable to write tests using words that are not in `L`.

76.5 Implementation and Testing

Based on the results for the previous steps of the design recipe, the implementation is displayed in Fig. 98. Run the tests, and make sure they all pass. As you may already suspect, `grammar-test` may easily be caught in an infinite recursion when used to test a `csg`. For this reason, its use with `csg`s is disabled in `FSM`.

77 A csg for Adding Expressions

Consider verifying valid adding expressions for numbers in unary notation. The language of valid adding expressions may be represented as follows:

$$L = \{AbBbAB \mid A,B \in i^*\}$$

For example, the following are valid arithmetic expressions:

```
iibibiii      bb      iiibiibiiiii
```

Respectively, they represent 2+1=3, 0+0=0, and 3+2=5. To verify that a given word is a valid adding expression, it must be in `L`. Observe that in `L`, context matters. A given expression is valid only if it ends with a number of `i`s that is equal to the number of `i`s before the second `b`. This suggests implementing a `csg` for `L` and using it to verify valid adding expressions.

77.1 Design Idea

The grammar may start by generating `AbBbE`. The `A` and `B` are used to nondeterministically generate the unary numbers in the sum and, for every `i` in these, to generate, `I`, a promise to generate an `i` for the result. Every `I` may be bubbled to the right until it reaches `E`. Upon reaching `E`, an `I` becomes an `i`. Nondeterministically, `E` generates `EMP`.

77.2 Name, Alphabet, and Syntactic Categories

A descriptive name for the grammar is `ADD-CSG`. The alphabet in {`b i`}.

The nonterminals represent syntactic categories to generate the elements of a valid adding expression in the proper context. The starting nonterminal, `S`, must generate three unary numbers separated by `b`s such that the sum

of the first two numbers is equal to the third number. It is documented as follows:

```
;;  S: generates words in i^nbi^mbi^ni^m
```

A nonterminal, `A`, is needed to generate a unary number. The generated number must contain zero or more `i`s. This nonterminal is documented as follows:

```
;; A: generates words in i^* and, for every i generated,
;;    a promise to generate a matching i for the result
```

Two nonterminals, `I` and `E`, are needed to generate the sum's result. The first represents the promise to generate an `i` in the proper context. The proper context is at the end of the word being generated. That is, `IE` generates an `i` for the sum of the first two numbers. These nonterminals are documented as follows:

```
;;  I: generates an i for the result in the context IE
;;  E: generates zero i or generates one i for the result
;;     in the context IE
```

77.3 Production Rules

`S` must generate three numbers separated by `b`s such that the third number is the sum of the first two numbers. The needed production rule is:

```
(S ,ARROW AbAbE)
```

The nonterminal `A` must generate an arbitrary number of `i`s and a matching promise for each to generate an `i` for the number representing the result. The needed production rules are:

```
(A ,ARROW ,EMP)
(A ,ARROW iIA)
```

To generate `i`s for the result number, the `I`s must be bubbled to the end of the partially generated word after the second `b` and before `E`. To do so, every `I` needs to skip over `i`s and `b`s to move right. The needed production rules are:

```
(Ii ,ARROW iI)
(Ib ,ARROW bI)
```

Finally, `E` nondeterministically decides to generate 0 `i`s or an `i` in the context `IE`. If generating an `i`, then `E` moves left to generate other `i`s for the result number. The needed production rules are:

```
(IE ,ARROW Ei)
(E ,ARROW ,EMP)
```

77.4 Tests

Tests are written using relatively short words that are in the language for the reasons mentioned before. Sample tests are:

```
;; Tests
(check-equal? (grammar-derive ADD-CSG2 '(b b))
              '(S -> AbAbE -> AbAb -> Abb -> bb))
(check-equal? (last (grammar-derive ADD-CSG2 '(b i b i)))
              'bibi)
(check-equal? (last (grammar-derive ADD-CSG2 '(i b b i)))
              'ibbi)
(check-equal?
  (last (grammar-derive ADD-CSG2 '(i i b i i b i i i i)))
  'iibiibiiii)
(check-equal?
  (last (grammar-derive ADD-CSG2 '(i i b i i i b i i i i i)))
  'iibiiibiiiii)
```

Be advised that the last test may take over 30 seconds to evaluate.

77.5 Implementation and Testing

Based on the results for the previous steps of the design recipe for grammars, `ADD-CSG`'s implementation is displayed in Fig. 99. Run the tests, and make sure they all pass.

78 Equivalence of csgs and tms

As we have just established, context-sensitive grammars can generate languages that are not context-free. We now explore the relationship between csgs and tms. As you know, grammars can be used to derive a word, `w`, in a language `L`. When $w \notin L$, grammars may provide no useful information because the search for a derivation may never terminate. What does this mean for the construction of a `tm` that simulates a grammar? Clearly, if a derivation is found by the grammar, its simulating `tm` halts. On the other hand, if a derivation is not found and the search continues forever, then the simulating `tm` shall never halt. This means that the simulating `tm` can only semi-decide `L`.

Given a `tm`, `M` that semi-decides `L`, how is a grammar to generate words in `L` be constructed? Intuitively, as done to construct a `cfg` from a `pda`, the grammar shall mimic any computation by `M` in reverse order. We shall assume that `M` always erases its tape before halting. That is, if `M` halts it halts in the following configuration: `(Y 1 `(,LM ,BLANK))`, where `Y` is the accept state. Any `tm` that does not satisfy this condition is easily transformed to satisfy it.

Fig. 99 The `csg` designed for valid adding expressions

```
;; L = AbBbAB | A,B in i*
;; Syntactic Categories
;;  S: generates words in i^nbi^mbi^ni^m
;;  A: generates words in i^* and, for every i generated, a promise
;;     to generate a matching i for the result
;;  I: generates an i for the result in the context IE
;;  E: generates zero i or one i for the result in the context IE
(define ADD-CSG2 (make-csg '(S A E I)
                           '(b i)
                           `((S ,ARROW AbAbE)
                             (A ,ARROW ,EMP)
                             (A ,ARROW iIA)
                             (Ii ,ARROW iI)
                             (Ib ,ARROW bI)
                             (IE ,ARROW Ei)
                             (E ,ARROW ,EMP))
                           'S))

;; Tests
(check-equal? (grammar-derive ADD-CSG2 '(b b))
              '(S -> AbAbE -> AbAb -> Abb -> bb))
(check-equal? (last (grammar-derive ADD-CSG2 '(b i b i)))
              'bibi)
(check-equal? (last (grammar-derive ADD-CSG2 '(i b b i)))
              'ibbi)
(check-equal? (last (grammar-derive ADD-CSG2 '(i i b i i b i i i i)))
              'iibiibiiii)
(check-equal? (last (grammar-derive ADD-CSG2 '(i i b i i i b i i i i i)))
              'iibiiibiiiii)
```

We shall only sketch the proof that `L` is generated by a `csg` if and only if `L` is semi-decided by a `tm`. That is, we shall not implement the constructors. There are two reasons for this. The first is that there are a series of implementation choices that must be made that get messy fairly quickly and that provide little insight into the equivalence of `csg`s and `tm`s. The second stems from the limitations that `FSM` imposes on us. The proof involves creating an `mttm`, and as discussed in Fig. 17, transforming an `mttm` into a standard `tm` may require an alphabet richer than any that can be generated using `FSM`.

Theorem 1 *L is generated by a* `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof (Sketch)

($\Rightarrow$) Assume L is generated by a `csg`.

Let G = (make-csg N Σ R S) be the grammar that generates L. We shall design a nondeterministic 3-tape `mttm`. Recall that, as discussed in Fig. 17, any `mttm` may be transformed into a `tm`.

Tape 0 contains, w, the input and is never mutated. Tape 1 is used to reconstruct a derivation for w starting with S. Therefore, M starts by writing S on tape 1. Tape 2 contains R and is never mutated.

M operates in steps as follows:

1. M nondeterministically picks a rule, y → x, to apply from tape 2 or chooses to match the contents of tape 2 with w on tape 0.
2. If a rule is chosen:
   a. M scans tape 1 and nondeterministically stops at a symbol.
   b. M matches y and tape 1 is mutated to replace y with x. Tape 1's contents is shifted as necessary to fit x.
3. If a rule is not chosen, M matches w and accepts or it runs forever

It is not difficult to see that M semi-decides L. M only halts if w is derived using G and, otherwise, runs forever. Recall that M only makes a nondeterministic choice if it leads to accept. If no such choice is possible it runs forever.

(⇐) Assume L is semi-decided by a `tm`.

Let M = (make-tm K Σ R S F Y) semi-decide L as outlined above. We shall construct a `csg`, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

- N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i $ua_iv$)) is represented by the symbol `LM`$ua_i$q`v`RM`. Observe that the state is to the immediate right of the symbol under the head and the right-end marker is at the end of the symbol.
- $\Sigma' = \Sigma$
- ∀k∈K and ∀a∈Σ, R' has rules built as follows:
  1. If ((Q a) (P b))∈R, where b∈Σ, then R has: Pb → aQ
  2. If ((Q a) (P `RIGHT`))∈R, then R' has the following rules:
     a. ∀b∈Σ R' has: abP → aQb
     b. To reverse extending the touched part of the tape to the right, R' has: a`BLANK`P → aQ`RM`
  3. If ((Q a) (P `LEFT`))∈R and a≠`BLANK`, then R' has: Pa → aQ
  4. If ((Q `BLANK`) (P `LEFT`)) the R' has the following rules:
     a. ∀ b∈Σ R' has: Pab → aQb
     b. To reverse erasing blanks, R' has: P`RM` → `BLANK`Q
- To start the derivation, R' has: S → `BLANKYRM` (i.e., the derivation starts where M halts)

- To erase the start state when the derivation is completed, R' has: LMBLANKs → EMP
- To erase the right-end marker when the derivation is done, R' has: RM → EMP □

It is not difficult to see that G only generates words that are accepted by M. When given a word, v, not in L, M runs forever. That is, it never reaches `(Y 1 `(,LM ,BLANK))`. Thus, G can never generate v.

1 The famous Turing Hacker is back to visit us. He claims that he has implemented a simpler csg for $L = a^n b^n c^n$:

```
(define anbncn-csg
  (make-csg '(S A B C G H I)
            '(a b c)
            `((S  ,ARROW ABCS)
              (S  ,ARROW ,EMP)
              (BA ,ARROW AB)
              (CA ,ARROW AC)
              (CB ,ARROW BC)
              (A  ,ARROW a)
              (B  ,ARROW b)
              (C  ,ARROW c))
            'S))

(check-equal? (grammar-derive anbncn-csg '())
              `(S -> ,EMP))
(check-equal? (grammar-derive anbncn-csg '(a a b b c c))
              '(S
                -> ABCS
                -> aBCS
                -> aBCABCS
                -> aBCABC
                -> aBCABc
                -> aBACBc
                -> aABCBc
                -> aAbCBc
                -> aabCBc
                -> aabBCc
                -> aabBcc
                -> aabbcc))
```

Is he correct? Justify your answer.

2 Design and implement a csg for L = {ww | w∈{a b}*}. Follow all the steps of the design recipe.

3 Design and implement a csg for L = {w | w has equal number of as, bs, and cs}. Follow all the steps of the design recipe.

4 Design and implement a csg for L = {w | w∈$a^n b^{2n} c^{3n}$. Follow all the steps of the design recipe.

5 Design and implement a csg for L = {ww^Rw | w∈{a b}*}. Follow all the steps of the design recipe.

Chapter 19
Church-Turing Thesis and Undecidability

Throughout this book, we have asked ourselves: What can and cannot be computed? On this journey, we have explored several computational models capable of deciding, semi-deciding, or generating languages. We have also seen how Turing machines can compute functions. In addition, we have seen that attempts to strengthen Turing machines have failed. It appears that we have reached the limit of what can be computed. Anything that can be computed may be computed by a Turing machine.

Recall that in Chap. 2, we asked one of the most fundamental questions in computer science: What is an algorithm? At the time, it is likely that you were ill-equipped to answer that question. Our quest to discover what can be computed has changed everything. You are now well-equipped to answer that question. An algorithm is a Turing machine. We need to be a bit more specific, however, because as we have seen, not all Turing machines are useful devices. Turing machines that decide a language or that compute a function are algorithms. Turing machines that may not halt are not considered algorithms. Stated in more pedestrian terms, a program that goes into an infinite recursion or an infinite loop is not the implementation of an algorithm.

A Turing machine that halts on all inputs is the formal notion of an algorithm. Nothing that cannot be implemented as a Turing machine that halts on all inputs is considered an algorithm. This principle is known as the *Church-Turing thesis*. It is coined a thesis, because there is no formal proof for it. In fact, this thesis cannot be proven because it is not a mathematical statement. It is only an assertion that states that a mathematical object known as a halting Turing machine is the same as our informal concept of an algorithm. In theory, it is possible for the Church-Turing thesis to be proven incorrect. For this, someone would have to propose a model of computation capable of performing a computation that a Turing machine cannot carry

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_19

out. Most computer scientists and mathematicians consider such an event highly unlikely.

As exciting as having a formal definition of an algorithm is, why else should we care about reaching this intellectual milestone? We ought to care because it opens the door to proving that there are computational problems for which a solution does not exist. A problem for which a solution does not exist is known as an *undecidable* or *unsolvable* problem. The existence of such problems is something that you may have already expected. In Chap. 4, for example, we argued that a finite representation for each language does not exist because there is a countable number of finite language representations and an uncountable number of languages to represent. Stated differently, the problem of computing a finite representation for every language is unsolvable. This does not mean that we can never compute a finite representation for a language. It means that there is no algorithm (i.e., halting Turing machine) that can always do so.

Pondering the limitations of language representations may not be an everyday primary concern in industrial computer science. Nonetheless, even industrial computer scientists may face an unsolvable problem, and then it becomes important to be able to recognize such problems and to prove they are undecidable. For instance, would it not be wonderful to have a program that analyzes a program we write and tells us if our program goes or does not go into an infinite recursion or an infinite loop? Indeed, such a program would be wonderful. This problem, as we shall see, is undecidable. If your boss ever asks you to write such a program, you will know it is an impossible task and be able to explain why it is an undecidable problem. Mind you, this does not mean that we can never prove that a given program never goes into an infinite recursion or loop. It means that we cannot always solve the problem. That is, there is no halting Turing machine (i.e., algorithm) that can always decide if a given program, given valid input, will halt.

79 The Halting Problem

We now proceed to study our first undecidable problem: the halting problem. Can a program be written in your favorite programming language that takes as input a program in said language and the input for it and that determines if the given program enters an infinite recursion or infinite loop on the given input? Stated differently, does the given program halt on the given input? You may imagine your program signature, purpose, and header as follows:

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

Let us assume that `halts?` can be implemented. That is, there is an algorithm to determine if a given program halts on a given input. The predicate, `halts?`, may be used an auxiliary function. We shall use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false. This ought to be reminiscent of a diagonalization proof. In essence, it is asking if `p` is not related to itself in terms of `halt?`. We can write such a function as follows:

```
;; program → Boolean
;; Purpose: Return #f if the given program does not
;;          halt on itself
(define (p-halts-on-p? p)
  (if (halts? p p)
      (p-halts-on-p? p)
      #f))
```

Why is `p-halts-on-p?` interesting? Consider calling `p-halts-on-p?` with `p-halts-on-p?` as input:

```
(p-halts-on-p? p-halts-on-p?)
```

When does `(p-halts-on-p? p-halts-on-p?)` return `#f` and halt? It does so when `(p-halts-on-p? p-halts-on-p?)` does not halt. This is clearly a contradiction. Therefore, we are forced to conclude that our initial assumption is wrong. The predicate `halts?` cannot be implemented. Thus, an algorithm to determine if on a given input a given program halts does not exist. Stated briefly, the halting problem is undecidable.

It is noteworthy that our discussion started by asking you to think about implementing a program in your favorite programming language. This programming language may be `FSM`. Therefore, the problem becomes to implement a Turing machine to determine if on a given input, a given Turing machine halts. Such a machine may be encoded as a `ctm`. This hypothetical machine must be a language recognizer for:

$$\texttt{L} = \{\texttt{(M w) | M=(make-tm K } \Sigma \texttt{ R S Y)} \wedge \texttt{w} \in \Sigma^* \wedge \text{M halts on w}\}$$

Based on the argument above, we can formally prove that `L` is undecidable.

Theorem 1 *The halting problem is undecidable.*

Proof

Assume L is decidable.

This means that there is a `ctm` (or `tm` if you like), `M`, that decides `L`. Consider a `tm`, `N`, that decides `L(N)`. Let `w` be an input word for `N`. Without loss of generality, assume the precondition for this machine is:

$$\texttt{PRE: tape = `(,LM BLANK w)} \wedge \texttt{ head position = 1}$$

We can easily edit N to create R that semidecides L(N). That is, R only differs from N by looping forever when N rejects w. Observe that we can build a ctm that decides L(N) using M as follows:

1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape

By assumption, this ctm returns 'accept if R halts on w and 'reject otherwise. If R halts then this means w∈L(N). If R does not halt, then this means w∉L(N). That is, the ctm uses M to decide if w is in L(N).

Consider the following language:

$$H = \{M \mid M \text{ is a tm encoding that halts when given itself as input}\}$$

A ctm that decides H gets as input M and M as input to itself. In essence, M is a tm capable of processing tms (much like an evaluator is a program that processes programs). Clearly, if H is decidable, then it is recursively enumerable. That is, we can write a program that will eventually print an arbitrary element in H. Furthermore, there exists a tm to decide its complement. This machine is obtained by flipping the roles of the accept and reject states.

The complement of H is:

$$\hat{H} = \{w \mid w \text{ is not a tm encoding} \vee w \text{ is a tm encoding that does not halt when given itself as input}\}$$

Observe, however, that $\hat{H}$ is not decidable. Let P be a tm encoding that semidecides $\hat{H}$. We can now ask if P is in $\hat{H}$. If P is in $\hat{H}$, then P does halt when given itself as input. However, P semidecides $\hat{H}$ and must halt if P does not halt when given itself as input. Stated differently, we have concluded P∈$\hat{H}$ if and only if P halts on itself. That is, P does not halt on itself if and only if P halts on itself. Clearly, this is a contradiction, and P cannot exist. If $\hat{H}$ is not semi-decidable, then it is also not decidable. Thus, our assumption that L is decidable is wrong, and L must be undecidable. □

1 The proof that the Halting problem is undecidable relies on decidable languages being closed under complement. Formally prove that decidable languages are closed under complement.

2 Formally prove that decidable languages are closed under union.

3 Formally prove that decidable languages are closed under concatenation.

4 Formally prove that decidable languages are closed under Kleene star.

5 Formally prove that decidable languages are closed under intersection.

80 Reduction Proofs

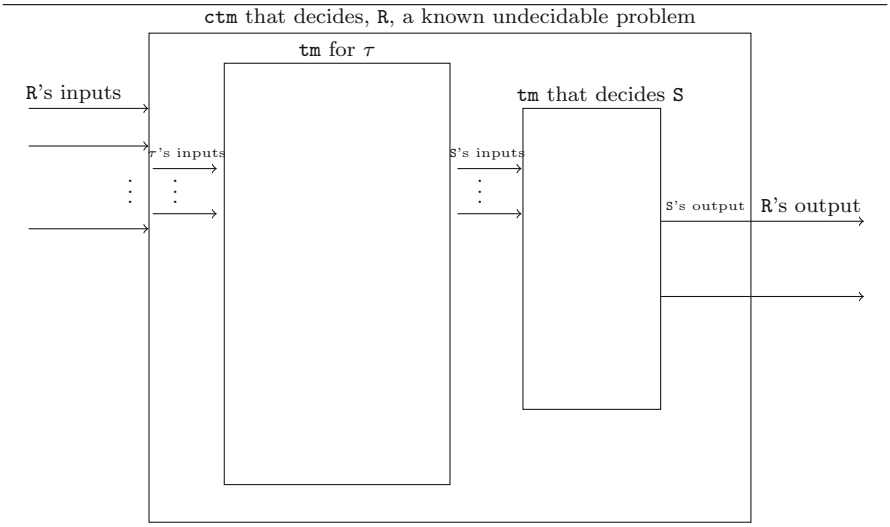
It is certainly remarkable that we have established that the halting problem is undecidable. Even more remarkable, perhaps, is that from the undecidability of the halting problem, the undecidability of many other problems follow. These results, however, do not follow the diagonalization argument used to prove the halting problem's undecidability. Instead, a reduction strategy is followed.

A *reduction proof* establishes that an undecidable language/problem, `L`, may be decided if some different language/problem, `L1`, is decidable. That is, the `tm` that decides `L1` may be used to build a `tm` that decides `L`. This, of course, is a contradiction because we know `L` is undecidable. Therefore, `L1` must also de undecidable.

It is important to note the direction of the reduction. When you try to prove that a language, `P`, is undecidable, assume that `P` is decidable, and establish a contradiction by showing how the `tm` that decides `P` may be used to decide a language that is known to be undecidable. In other words, use the `tm` that decides `P` as a subroutine in a `ctm` that decides a language known to be undecidable. Observe that if the reduction is done in the wrong direction, nothing new is established. That is, showing that `P` is decidable using a hypothetical `tm` for an undecidable problem does not establish that `P` is undecidable. It simply establishes that if a undecidable problem were decidable, then `P` is also decidable. There is no contradiction. Granted, the `tm` designed to decide `P` cannot exist. This, however, does not mean `P` is undecidable. It simply means that we have not found an algorithm to decide `P` that does not involve the use of hypothetical `tm`s that cannot exist.

More formally, let `R` and `S` be two problems. A reduction from `R` to `S` is a transformation function, τ, such that $\texttt{x} \in \texttt{R} \Leftrightarrow \tau(\texttt{x}) \in \texttt{S}$ and $\texttt{x} \notin \texttt{R} \Leftrightarrow \tau(\texttt{x}) \notin \texttt{S}$. You may think of τ as a function that is computable by a `tm` that transforms a subset of inputs for `R` to inputs for `S` in such a manner that the answer returned by `S` means that we know the answer that `R` ought to return. Figure 100 displays a graphical representation of such a reduction for an undecidable problem `R`. A `ctm` to decide `R` is built using two auxiliary `tm`s (or `ctm`s). The first is for τ. This machine converts a subset of `R`'s input to serve as inputs to the second auxiliary machine. The second is a hypothetical auxiliary machine that decides `S`. The output of `S` is used to return an

Fig. 100 A graphical template for a reduction proof from R to S



answer for R. The following theorem establishes the correct employment of a reduction.

Theorem 2 *R is undecidable* $\wedge \exists$ τ = *a reduction function from R to S* $\Rightarrow$ *S is undecidable.*

Proof

Assume M_S decides S and T computes τ.

The ctm T M_S decides R. This is a contradiction. Therefore, S is undecidable.

Hopefully, it is now clear that if R is undecidable, then a tm to decide S cannot exist. If such a machine existed, then R would be decidable. As with the halting problem, the proof above does not imply that there does not exist an instance for which R can be solved. There may be many possible inputs for which a solution may be found by other means (akin to proving that a given while-loop or recursion halts). There is, however, no general algorithm to decide R.

81 Undecidable Problems About Turing Machines

Alan Turing penned one of the great intellectual achievements of the 20th when in 1936, he proved that the halting problem was undecidable. It paved the way to prove that other problems are undecidable using reduction proofs.

In this section, we shall explore undecidable problems about Turing machines. For each problem, S, the strategy shall be the same. We identify an undecidable problem, R, that becomes decidable if S were decidable, and we describe, τ, the reduction function.

81.1 M Halts on EMP

Does a tm, M, halt on the empty tape? That is, does M halt when it starts with an empty tape? This problem is undecidable as established by the following theorem.

Theorem 3 *Given a tm, M, determining if M halts on EMP is undecidable.*

Proof

Assume S decides if M halts on EMP. That is, S decides the following language:

L = {N | N is a tm or ctm that halts on EMP}

We shall use S to build a ctm, R, that decides the halting problem.

The inputs for R are M and a word w. The reduction machine for τ builds a tm, M', that operates as follows:

1. M'’s starting configuration is: (S 1 `(,LM ,BLANK))
2. M' writes w on the tape
3. Moves the head to the first blank to the left (i.e., the first blank on the tape)
4. Simulates M

Stated using the graphical ctm notation, for w = $a_1 a_2 \ldots a_n$ M' is:

R a_1 R a_2...R a_n FBL M

M' is given as input to S. If S accepts, then M halts on w, and the R's output is accept. Otherwise, if S rejects, then M does not halt on w, and the R's output is reject. Given that the halting problem is undecidable, we have a contradiction, and therefore, S cannot exist. □

81.2 There Exists a Word for Which M Halts

We now turn our attention to the problem of determining if there exists any word for which a given tm, M, halts. This problem is undecidable as established by the following theorem.

Theorem 4 *Given a* `tm`*, M, determining if there exists a word for which* `M` *halts is undecidable.*

Proof

Assume `S` decides if there exists any word for which a given `tm` halts. That is, S decides the following language:

```
L = {N | N is a tm or ctm such that there exists a word
         for which N halts}
```

We shall use `S` to build a `ctm`, `R`, which decides if M halts on `EMP`. This language is proven undecidable in Sect. 81.1.

The input to `R` is, M, an arbitrary `tm`. The reduction machine for τ builds a `tm`, `M'`, which operates as follows:

```
1. Erases its input
2. Simulates M
```

`M'` is given as input to `S`. If `S` accepts, this means that `M'` halts on some word if and only if it halts on all words. Therefore, M halts on `EMP`. Thus, `R` ought to accept. Otherwise, if `S` rejects, this means that `M'` does not halt on any word if and only if it does not halt on all words. Therefore, M does not halt on `EMP`. Thus, `R` ought to reject. Given that determining if a `tm` halts on `EMP` is undecidable, we have a contradiction, and therefore, `S` cannot exist. □

81.3 Does M Ever Reach Q Given w?

Given an arbitrary `tm`, `M`, a `Q∈(sm-states M)`, and a word `w`, we wish to determine if `M` ever reaches `Q` during its computation on `w`. This problem is undecidable as established by the following theorem.

Theorem 5 *Given a* `tm`*, M, a state, Q∈(sm-states M), and an input word w, determining if M reaches Q when given w as input is undecidable.*

Proof

Assume `S` decides if M ever reaches Q when given w as input. That is, S decides the following language of triples:

```
L = {(N Q w) | N is a tm or ctm such that N reaches Q when
               given w as input}
```

We shall use `S` to build a `ctm`, `R`, which decides the halting problem.

The input to `R` is, M, an arbitrary `tm`, and, w, an arbitrary word. The reduction machine for τ builds a `tm`, `M'`, which operates as follows:

1. M' has a single final state, H, that is not a state in M.
2. Simulates M on w and moves to H if M halts

M', H, and w are given as input to S. If S accepts, this means that M' reaches its final state. This only happens if M halts on w. Therefore, R ought to accept. If S rejects, then this means that M' does not reach its final state. This only happens if M does not halt on w. Therefore, R ought to reject. Given that the halting problem is undecidable, we have a contradiction, and therefore, S cannot exist. □

6 Prove that determining if a given tm halts on all inputs is undecidable.

7 Prove that determining if two given tms halt on the same input is undecidable. Hint: Build on the solution to the previous problem.

8 Prove that determining if a given tm applied to EMP writes a given symbol a is undecidable.

9 Prove that determining if there is any word for which two given tms halt is undecidable.

10 Prove that determining if the language semidecided by a given tm is finite is undecidable.

82 Undecidable Problems About Grammars

Undecidable problems also exist for grammars. We shall now study undecidable problems about grammars. The same reduction strategy used for undecidable tm problems is employed.

82.1 Determine if w Is in the Language of a Grammar

Given a csg, G, consider determining if w∈L(G). It would be very useful to have an algorithm to decide this problem. This problem, however, is undecidable as established by the following theorem.

Theorem 6 *Given a* `csg`*, G, determining if a word, w, is in the language generated by G is undecidable.*

Proof

Assume `S` decides if w∈L(G). That is, S decides the following language of doubles:

`L = {(G w) | G is a csg and w∈L(G)}`

We shall use `S` to build a `ctm`, `R`, which decides the halting problem.

The input to `R` is, M, an arbitrary `tm`, and, w, an arbitrary word. The reduction machine for τ builds a `csg`, `G'`, using M and the construction algorithm sketched in Sect. 78. `L(G')` is the language semidecided by M.

`G'` and w are given as input to `S`. If `S` accepts, then we know that `G'` generates w. This means M halts and accepts w. Therefore, `R` ought to accept. If `S` rejects, then we know that `G'` does not generate w. This means M does not halt on w. Therefore, `R` ought to reject. Given that the halting problem is undecidable, we have a contradiction, and therefore, `S` cannot exist. □

82.2 Is `L(G)` Empty?

Theorem 7 *Given a* `csg`*, G, determining if L(G)=∅ is undecidable.*

Proof

Assume `S` decides if L(G)=∅. That is, S decides the following language of doubles:

`L = {G | G is a csg and L(G)=∅}`

We shall use `S` to build a `ctm`, `R`, which decides if there is any word for which a `tm` halts. This `tm` problem is proven undecidable in Sect. 81.2.

The input to `R` is, M, an arbitrary `tm`. The reduction machine for τ builds a `csg`, `G'`, for M using the construction algorithm sketched in Sect. 78. `L(G)'` is the language semidecided by M.

`G'` is given as input to `S`. If `S` accepts, then we know that `L(G)'`=∅. This means L(M) is empty. Therefore, `R` ought to reject. If `S` rejects, then we know that `L(G)'`≠∅. This means L(M) is not empty. Therefore, `R` ought to accept. Given that determining if there is any word for which a `tm` halts is undecidable, we have a contradiction, and therefore, `S` cannot exist. □

11 Given a `csg`, `G`, prove that determining if `EMP`$\in$`L(G)` is undecidable.

12 Given two `csg`, `G` and `J`, prove that determining if `L(G)=L(J)` is undecidable.

Chapter 20
Complexity

One of the principal threads in this book is nondeterminism. We learned in Sect. 28 that, computationally speaking, deterministic and nondeterministic finite-state machines are equivalent. That is, given a nondeterministic finite-state machine that decides a language `L`, we can build a deterministic finite-state machine that decides `L`. We learned in Sect. 57 that, computationally speaking, nondeterministic pushdown automata are more powerful than deterministic pushdown automata. That is, there are context-free languages decided by nondeterministic pushdown automata that cannot be decided by deterministic pushdown automata.

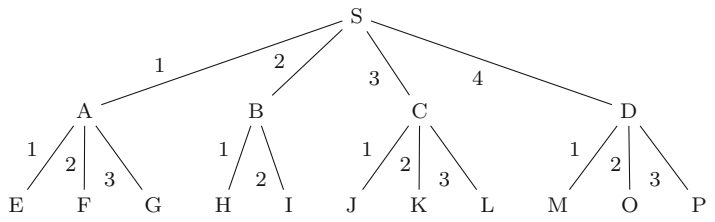
In Sect. 59, we learned that Turing machines may be deterministic. Later, in Sect. 61, we learned to design nondeterministic Turing machines. We also learned, in Chap. 17, that proposed extensions for Turing machines result in machines that may be simulated by a standard Turing machine. Furthermore, Turing machines decide regular languages (see Sect. 62), and a two-tape Turing machine may be used to simulate any pushdown automata (see exercise at the end of Sect. 71). The natural question that arises is whether or not deterministic and nondeterministic Turing machines, computationally speaking, are equivalent. If so, deterministic Turing machines are powerful enough to simulate nondeterministic Turing machines, pushdown automata, and finite-state machines.

83 Equivalence of Deterministic and Nondeterministic Turing Machines

Nondeterminism in `tm`s, on the surface, seems to be a powerful feature that may not be eliminated. In fact, however, a nondeterministic `tm` may be simulated by a deterministic `tm`. Eliminating nondeterminism in a `tm` is not

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_20

Fig. 101 Computation tree



done by cleverly encoding information as described in Chap. 17 to establish the equivalence of Turing machine extensions. There is no known algorithm using such an approach to eliminate nondeterminism from a `tm`.

Instead of cleverly encoding information, we shall develop an algorithm for a deterministic `tm` to systematically simulate all the possible steps that a nondeterministic `tm` may take – even those that do not lead to the machine halting or accepting. In essence, this algorithm performs an exhaustive search of all the possible transitions, the nondeterministic `tm` would perform if it did not have the ability to sense the right transition to apply when a nondeterministic choice must be made.

83.1 Design Idea

Given a nondeterministic `tm`, `N`, that semi-decides a language, `L`, we need to build a deterministic `tm`, `D`, that semi-decides `L`. Without loss of generality, we shall assume that `N` always has at least one rule that may be used for any configuration that is not in a final state and that it only halts if it accepts. At each step, `N` chooses among all applicable rules to advance a computation. `N` has the power to sense which is the right rule to apply. `D` cannot sense which is the right rule to apply. Therefore, it must simulate a computation for each possible choice as it searches for any computation that accepts the input word. The search space may be visualized as a computation tree as displayed in Fig. 101. Each node represents a configuration for `N`. The edges represent a rule application (i.e., a step). For example, in one step, `N` can change from configuration `C` to configuration `K`. The root of the tree is `N`'s starting configuration.

If a node has more than one child, then `N` nondeterministically chooses the path to follow. The deterministic machine must simulate all paths. To this end, we need a systematic way to represent a computation (i.e., a path from the root to any node). A possible representation is a word of encoded numbers. Each number represents a choice made by `N`. The first number represents the choice made at step 1, the second number represents the choice

made at step 2, and so on. This requires knowing the maximum number of rules that may be used on any configuration. Let us denote this number by `r`. This means the maximum number of children any node may have in a computation tree is `r`. For the purposes of this discussion, assume `r` is 4. Observe that in Fig. 101, the maximum number of children is 4 (for node `S`). The computation that takes `N` from `S` to `D` to `M` is the encoding for `(4 1)`. This means that `N` chose the fourth applicable rule for step 1 and the first applicable rule for step 2. It is not difficult to see that every path in a computation tree may be encoded in this manner regardless of the tree's height. The edges in Fig. 101 are labeled with such choice numbers.

To systematically explore each computation path of varying lengths, the deterministic machine can perform a breadth-first search of the computation tree exploring shorter paths before longer paths. That is, the deterministic machine first explores all paths of length 0, then all paths of length 1, then all paths of length 2, and so on. The machine halts if it finds a path that takes `N` to a final state. The key to making this process systematic is to define an increment function for a word (of encoded numbers) that represents a computation path. We coin this word the computation number. Every member of the computation number is an element in `[0..r]`. Assume that there is a blank before such a word on the tape that contains it. The increment function may be described as follows:

1. If the rightmost number is less than `r`, add 1 to it.
2. If the rightmost number is `r`, make it 1, and propagate a 1 to the left.
3. During left propagation, if a number is less than `r`, add 1 to it. If the number is `r`, make it a 1, and propagate left. If the blank is read, then make the blank a 1, and shift the number one space to the right.

To illustrate how the increment function works, let us consider the first 12 increments and the computation paths in Fig. 101. The process starts with 1:

```
(1) represents the computation S ⊢ A
(2) represents the computation S ⊢ B
(3) represents the computation S ⊢ C
(4) represents the computation S ⊢ D
```

At this point, the number equals `r`, and the increment must propagate left:

```
(1 1) represents the computation S ⊢ A ⊢ E
(1 2) represents the computation S ⊢ A ⊢ F
(1 3) represents the computation S ⊢ A ⊢ G
(1 4) represents a computation that does not exist
```

Once again, the number equals `r`, and the increment must propagate left:

```
(2 1) represents the computation S ⊢ B ⊢ H
(2 2) represents the computation S ⊢ B ⊢ I
(2 3) represents a computation that does not exist
(2 4) represents a computation that does not exist
```

We can now outline an algorithm for a tm to systematically search a computation tree. It shall use an iterative deepening strategy and have three main values (more values are needed in an actual implementation), the input word w, N's current configuration C, and the encoded number, I, representing the current computation performed. The machine's operation is outlined as follows:

```
1. If N's starting state is a final state, then halt.
2. Start with N's starting configuration and I = (1).
3. Perform the computation encoded by I if possible.
   If not possible, ignore the rest of I and return C.
4. Check the configuration returned by Step 3. If it is
   in one of N's halting  states, then halt. Otherwise,
   increment I, restore N's starting configuration, and
   go to step 3.
```

In essence, the deterministic machine performs a computation of length d and checks if N would have reached a halting state. If so, the deterministic machine halts. Otherwise, it restarts from N's starting configuration and performs the next computation, if any, of length d or length d + 1. Given how I is incremented, a computation of length d + 1 is only performed after all computations of length less than or equal to d are performed.

83.2 Correctness

The constructive proof shall focus on a tm that semidecides a language. The construction for a tm that decides a language or that computes a function is very similar. To avoid falling into the Turing pit, we shall not pin down all the low-level details of the transitions. Instead, we shall sketch the proof with enough detail to convince any reader that a deterministic tm can carry out the simulation of the given nondeterministic tm.

Theorem 1 *A nondeterministic tm semidecides $L \Rightarrow \exists$ a deterministic tm that semidecides L*

Proof (Sketch)

Assume N is a nondeterministic tm that semidecides L.

An mttm with five main tapes shall simulate N. The purpose of each tape is described as follows:

```
Tape 0: Contains the input word, w, and is never mutated
Tape 1: Used to simulate N by recording N's configuration
Tape 2: Stores, I, the next computation number
```

```
Tape 3: Stores N's rules
Tape 4: Stores the rules applicable to the current
        configuration on Tape 1
```

More tapes may be needed to manage low-level details, but we shall not concern ourselves with these details at this time. Clearly, an `mttm` may have as many tapes as needed.

The `mttm` starts with the input word, w, on tape 0 and operates as follows:

1. Write the encoding for the computation number '(1) onto tape 3
2. Write N's starting configuration on tape 2 using w on tape 1
3. Read the rightmost unprocessed encoded element, k, in the computation number on tape 3
4. Extract the rules from tape 3 that may be used on the configuration on tape 1, and copy them to tape 4
5. Apply the kth rule on tape 4 to the configuration on tape 1, and mutate tape 1 to store this new configuration. If no such rule exists, go to the next step.
6. Check if tape 1 contains a halting configuration
   a. If so, halt
   b. Otherwise, increment the computation number on tape 3, and go to step 2

Although many low-level details (e.g., how to extract the kth rule on tape 4 and extracting the computation number's rightmost unprocessed element) are left unspecified, it is not difficult to see that every step taken by the sketched `mttm` is deterministic. In Sect. 72, we outlined how to convert an `mttm` into a standard `tm`. This conversion does not introduce nondeterministic operations that are not present in the given machine. Therefore, the result of transforming the proposed `mttm` is a deterministic `tm`.

1 Sketch a proof demonstrating that if a nondeterministic Turing machine, `N`, decides a language, `L`, then there exists a deterministic Turing machine that decides `L`.

2 Sketch a proof demonstrating that if a nondeterministic Turing machine, `N`, computes a function, `f`, then there exists a deterministic Turing machine that computes `f`.

84 Does Solvable Mean a Practical Solution?

We have studied problems that can be solved using a `tm`s, and we have studied problems that cannot be solved using a `tm`. Therefore, we categorize problems as: solvable or unsolvable. There is nothing we can do about the unsolvable problems. Therefore, as problem-solvers and computer scientists, we are delighted that there are many solvable problems we can put our skills to use. The question now becomes whether or not the solution to a solvable problem is practical.

To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (`TSP`). Given `k` cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way. This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

The second step is straightforward to solve. For each itinerary, sum the distances between adjacent cities, and return the path with the smallest sum. Summing the distances for each itinerary is done in a number of steps proportional to `k`: `k` distances are summed. This part of the algorithm is quite manageable. The question becomes how many times must a manageable sum be computed. This depends on the first step. Assume that the starting city is `A`. Each valid itinerary looks as follows:

```
                City: A ___ ___ ___ ···  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

The number of cities between the `A`s is `k-1`. For the first of these cities, we have `k-1` cities to choose from; for the second, we have `k-2` cities to choose from and so forth until the `k-1` city for which we only have one choice. This means that the number of different itineraries is equal to `(k - 1)!`. This may look innocent enough, but think carefully about what this means. If our hypothetical salesman had to visit six cities, then number of possible itineraries is $5! = 120$. A computer can quickly check 120 itineraries and select the shortest one. Consider, however, the possibility that our hypothetical salesman must travel to 50 cities. In this case, the number of possible itineraries is 49![15] Can these many itineraries be processed in a reasonable amount of time? If 10 billion itineraries could be processed per second,[16] then it would take about 1928849137602319764308257747721002590333437440 years to find the shortest itinerary.

[15] $49! = 608281864034267560872252163321295376887552831379210240000000000$.

[16] This is faster than any computer in the foreseeable future.

Clearly, even for a modest value for `k`, finding the solution takes too long. That is, the algorithm solves the problem, but it is not practical. No salesman could wait that many years for an itinerary.

85 The Class $\mathcal{P}$

We now understand that having a solution to a problem does not mean that the solution is practical. For example, the number of operations performed to solve the traveling salesman problem is proportional to `(k - 1)!`, which grows faster than 2^k. That is, the number of operations that need to be performed in the worst case grows exponentially as the size of the input grows. In other words, the growth function is an exponential function. Such growth renders the solution impractical. In contrast, other algorithms are considered practical. For example, consider the problem of finding a pattern (or lack of a pattern) in a given word from Sect. 25.1. Recall that in the worst case, the entire word must be traversed. This means that the number of operations performed is proportional to, `n`, the length of the word. That is, the number of operations equals `k * n`, where `k` is a constant of proportionality. Observe that `k * n` does not grow exponentially. The growth function is a polynomial.

85.1 Defining Practical Solutions

We need a way to characterize solutions that are practical. That is, solutions that exhibit polynomial growth on the number of operations performed as the input's size grows. This eliminates deterministic `tm`s that simulate nondeterministic `tm`s. This follows from observing that the number of computation paths in a computation tree (like that one displayed in Fig. 101) grows exponentially. This leaves us with (a subset of) deterministic Turing machines that directly solve a problem (i.e., not simulating a nondeterministic machine). Let `n` be the size of the input, and let `k` be a non-negative constant. We define the set of Turing machines (i.e., algorithms) that are practical (i.e., solve a problem in a polynomial number of steps) as:

$$\mathcal{P} = \{\texttt{M} \mid \texttt{M is a deterministic Turing machine that decides a language or computes a function in a number of steps proportional to } \texttt{n}^k\}$$

We say that any problem solvable by a $\texttt{tm} \in \mathcal{P}$ is *tractable*.

A word of caution is opportune at this point. Not every practical algorithm (i.e., in $\mathcal{P}$) means it is suitable for use in everyday computing. Consider, for example, a deterministic `tm` whose number of steps is proportional to $\texttt{n}^{10}$. For a small input size of 10, the number of operations is proportional to 10^{10}

Fig. 102 Constructor for the complement of a deterministic `tm`

```
;; tm-language-recognizer → tm-language-recognizer
;; Purpose: Build a deterministic tm-language-recognizer for the
;;          complement of the language decided by the given
;;          deterministic tm-language-recognizer
;; Assumption: Given tm's final states are Y and N
;;             and Y is the accepting state
(define (dtm4L->dtm4notL M)
  (make-tm (sm-states M)
           (sm-sigma M)
           (sm-rules M)
           (sm-start M)
           (sm-finals M)
           'N))

;; L = a*
;; PRE: tape = LMw_ AND i = 0
(define a* (make-tm '(S Y N)
                    `(a b)
                    `(((S a) (S ,RIGHT))
                      ((S b) (N b))
                      ((S ,BLANK) (Y ,BLANK)))
                    'S
                    '(Y N)
                    'Y))


(define Not-a* (dtm4L->dtm4notL a*))

(check-equal? (sm-apply Not-a* `(,LM a a a b a a)) 'accept)
(check-equal? (sm-apply Not-a* `(,LM b a a)) 'accept)
(check-equal? (sm-apply Not-a* `(,LM)) 'reject)
(check-equal? (sm-apply Not-a* `(,LM a a a)) 'reject)
```

= 10000000000. For a modest input size of 100, the number of operations is proportional to 10^{100}. It is difficult to see how such an algorithm can be practical for other than the smallest instances of the problem solved. Thankfully, most algorithms of interest to industry programmers are solved in a number of steps proportional to n^3 or less.

85.2 Closure Under Complement

We can prove properties of $\mathcal{P}$ just like we proved properties for other classes of languages. For instance, $\mathcal{P}$ is closed under complement.

Theorem 2 *$\mathcal{P}$ is closed under complement.*

Proof

Let M = (make-tm K Σ R S '(Y N) 'Y) be deterministic, and decide a language L in polynomial time. This means M$\in\mathcal{P}$. The interpretation of the final states is as you may expect: Y is for accept, and N is for reject.

From M, we can build a deterministic Turing machine, $\bar{\text{M}}$, to decide L's complement in polynomial time. Every word accepted by M ought to be rejected by $\bar{\text{M}}$, and every word rejected by M ought to be accepted by $\bar{\text{M}}$. To achieve this, $\bar{\text{M}}$ is the same as M, except that the roles of Y and N are swapped. In this manner, $\bar{\text{M}}$ only accepts if M would have rejected and vice versa. □

An implementation of the constructor described in proof is displayed in Fig. 102.

86 The 2-Satisfiability Problem

A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem. The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable (i.e., is it possible for the formula to evaluate to true). Consider, for example, the following Boolean formula in conjunctive normal form:

```
(and (or x y) (not x) (or y z))
```

This formula is satisfiable. To establish this, we may make `x = #f`. The formula simplifies to:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

We may now make `y = #f`. The formula simplifies to:

```
(and #t (or #f z)
z
```

Clearly, making `z = #t` establishes the formula is satisfiable. In contrast, the following formula is not satisfiable:

```
(and (or x y z)
     (or (not x) (not y) (not z))
     (or x (not y))
     (or y (not z))
     (or (not x) z))
```

Let us establish this. The first clause, `(or x y z)`, is satisfied if any variable is true. The second clause, `(or (not x) (not y) (not z))`, is satisfied if any variable is false. So far, we know that at least one variable must be true and at least one variable must be false. The remaining conjunction

```
(and (or x (not y)) (or y (not z)) (or (not x) z))
```

is only satisfiable if the value of all three variables is the same. Thus, formula is not satisfiable.

We shall now explore how to solve a simplified version of the Boolean satisfiability problem called the 2-satisfiability problem. In this simplified version every clause of a formula in disjunctive normal form is either a singleton or a 2-disjunction. A singleton is either a variable or the complement of a variable. A variable is a symbol. The complement of a variable is a `not`-expression with a symbol. A 2-disjunction is an `or`-expression with two singletons. The following are valid formulae for the 2-satisfiability problem:

```
'(and x (or (not y) (not x)) (or (not z) y) y)
'(and (or x1 x2) x1 (or (not x1) (not x2)) (or (not x1) x2))
'(and (or a b) a (or (not a) (not c)) (not c))
'(and (or x1 x2)
      (or (not x1) (not x2))
      (or x2 x3)
      (or (not x1) (not x3)))
```

The solution presented shall draw upon our knowledge of grammars to represent input formulae given by the user. In addition, the grammar for input formulae is used to develop a formula parser. A parser is a function that transforms data into a representation that makes it easier to write programs that process said data. Specifically, parsers produce parse trees that eliminate the need for superfluous information like parentheses and keywords (e.g., `not` and `or`).

86.1 Representing Input Formulae

We need to decide how a user inputs a formula. The chosen representation ought to be natural for the user and avoid being too cumbersome. For our purposes, we shall adopt that an input formula is written as an `and`-expression with an arbitrary number of input clauses. That is, a list that contains the keyword `and` and a list of input clauses. An input clause is either an input singleton or an input 2-disjunction. An input singleton is either a symbol (representing a variable) or a `not`-expression containing a symbol (i.e., a list containing the keyword `not` and a symbol representing a variable). Finally, an input 2 disjunction is an `or`-expression with two input singletons (i.e., a list containing the keyword `or` and two input singletons.

Fig. 103 Context-free grammar for the 2-satisfiability problem formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

The data definitions above are likely a bit cumbersome for any user to remember. To help a user understand the data definitions, they may be concisely described using a context-free grammar as displayed in Fig. 103. The grammar informs any user how to input a valid formula for the 2-satisfiability problem. Such a grammar is known as a *concrete grammar*. It specifies everything the user must type to input a valid formula. Any user familiar with `FSM` syntax is very likely to find inputting formulae for the 2-satisfiability problem quite natural.

86.2 Parsing Input Formulae

Manipulating formulae written in concrete syntax is cumbersome, error-prone, and unnecessarily complex. This stems from the fact that many superfluous elements, like parentheses and keywords, must be processed. Such superfluous elements force programmers to use low-level functions, such as `car` and `cdr`, to manipulate formulae. As programmers, we prefer to write formula-processing functions at a higher level of abstraction. That is, we would like to manipulate the elements of a formula instead of, for example, manipulating lists to skip parentheses and keywords.

This may be achieved by transforming concrete syntax into *abstract syntax*. In abstract syntax, superfluous elements are eliminated by transforming an input formula into the parse tree representing its derivation using the grammar in Fig. 103. When written in abstract syntax, a formula does not retain the superfluous elements like parentheses and keywords. A program that converts concrete syntax into abstract syntax is called a *parser*.

To write a parser, we need a concrete grammar (which we have) and a representation for the syntactic categories of the concrete grammar without the superfluous elements. That is, we need a grammar for the abstract representation that is manupalted by a program. With this goal in mind, a *formula* is

represented a list of clauses. Do not confuse formula with iformula. A formula represents abstract syntax. We define formula as follows:

```
;; A formula → '()
;;           → (cons clause formula)
```

Observe that `and` is not part of the representation of a formula. We interpret a list of clauses as the conjunction of said clauses.

A clause is defined as follows:

```
;; clause → singleton
;;        → 2disjunction
```

These rules inform us that there two varieties of clauses. A `singleton` represents a variable or the complement of a variable. A `2disjunction` represents the disjunction of two singletons.

A singleton is defined as follows:

```
;; singleton → (var symbol)
;;           → (notvar symbol)

(struct var (symb) #:transparent)
(struct notvar (symb) #:transparent)
```

Each variety of singleton is a structure containing a symbol representing a variable. Observe that the representation for a variable's complement does not store the keyword `not`. Structure definitions contain inside parentheses the keyword `struct`, the name of the structure, and a list of the symbols (one for each field). In the two structure definitions above, each has a single field named `symb`. The constructor for a defined structure is its name, and it expects as input a value for each field. For example, `(notvar 'x)` builds the representation for `x`'s complement. When a structure is defined, a selector function for each field is created. A selector function name always follows the same pattern, <structure name>-<field-name>. For instance, consider the following definition:

```
(define VAR1 (var 'k))
```

To extract the symbol representing the variable stored in `VAR1`, `(var-symb VAR1)` is used. Finally, `#:transparent` makes the values stored in a structure visible. This illustrated with the following interaction:

```
> VAR1
(var 'k)
```

Without `#:transparent`, the value of `symb` inside any `var`-structure would not be visible.

Finally, a disjunction is represented as follows:

```
;; disjunction → (2disjunction singleton singleton)

(struct 2disjunction (s1 s2) #:transparent)
```

Fig. 104 The iformula parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)

  ;; icl → clause
  ;; Purpose: Parse the given icl
  (define (parse-iclause an-icl) ...)

 (map parse-iclause (rest an-iformula)))

(check-equal? (parse-iformula '(and x1 x2))
              (list (var 'x1) (var 'x2)))
(check-equal? (parse-iformula '(and (not i) j))
              (list (notvar 'i) (var 'j)))
(check-equal? (parse-iformula
               '(and (or a b) (or (not x) b)))
              (list
               (2disjunction (var 'a) (var 'b))
               (2disjunction (notvar 'x) (var 'b))))
(check-equal?
 (parse-iformula
  '(and (or (not n) m) r (or s (not t)) (not y)))
 (list
  (2disjunction (notvar 'n) (var 'm))
  (var 'r)
  (2disjunction (var 's) (notvar 't))
  (notvar 'y)))
```

Observe that the keyword `or` is not included as part of a disjunction. We interpret that a disjunction represents an `or`-expression with two singletons. As you expect, the selector functions for this structure are `2disjunction-s1` and `2disjunction-s2`.

86.3 The Formula Parser

The `iformula` parser takes as input an `iformula` and returns a `formula`. For code development, we use an `iformula`'s structure as defined by the grammar in Fig. 103. This transformation is achieved by parsing each of the clauses in the given input. This may be done by applying `map` to a function to parse an `iclause` and the rest of the given `iformula` (to skip the keyword `and`). The result of this design, including tests, is displayed in Fig. 104.

The auxiliary function `parse-iclause` creates a parse tree for the given `iclause`. It dispatches on the subtype of the given `iclause`: `isingleton` or `i2disjunction`. An auxiliary predicate `isingleton?` determines if a given `iclause` is an `isingleton` by testing if it is a symbol or a list whose first

Fig. 105 The iclause parser

```
;; icl → clause
;; Purpose: Parse the given icl
(define (parse-iclause an-icl)
  ;; isingleton → singleton
  ;; Purpose: Parse the given isingleton
  (define (parse-isingleton an-is)
    (if (symbol? an-is)
        (var an-is)
        (notvar (second an-is))))

  ;; idisjunction → disjunction
  ;; Parse: Parse the given idisjunction
  (define (parse-i2disjunction an-id)
    (2disjunction (parse-isingleton (first an-id))
                  (parse-isingleton (second an-id))))

  ;; iclause → Boolean
  ;; Purpose: Determine if the given iclause is an isingleton
  (define (isingleton? an-icl)
    (or (symbol? an-icl)
        (eq? (first an-icl) 'not)))

  (if (isingleton? an-icl)
      (parse-isingleton an-icl)
      (parse-i2disjunction (rest an-icl))))
```

element is `'not`. To parse an `isingleton`, a `var` structure is constructed if given a symbol. Otherwise, a `notvar` structure is constructed with the second element of the given list (thus, not storing the keyword `not`). To parse an `idisjunction`, a `2disjunction` structure is constructed with the results of parsing its two `singleton`s. The result of the design is displayed in Fig. 105.

86.4 The 2-Satisfiability Solver

The next problem to solve is finding variable assignments that satisfy a given `formula`. The coding part of our job is simplified given that we are processing parse trees and not `iformula` as entered by a user.

86.4.1 Design Idea

The 2-satisfiability solver, `2-satisfiability`, searches for a set of assignments to satisfy a given `formula`. It returns `'accept` if variables may be assigned values that satisfy the formula. Otherwise, `'reject` is returned.

The variable assignment search is delegated to an auxiliary function, `solve`, which employs an accumulator to store the current variable assignments in a list. This accumulator starts as the empty list. The accumulator invariant is that the accumulator contains the variable bindings made so far to satisfy the formula. The function first checks if any variable assignment is possible. If the given formula is empty, then all needed variable assignments have been made, and the accumulator is returned. If the formula has no solution, then the empty list is returned. A formula has no solution when a variable and its complement are both clauses in the formula. An auxiliary function, `has-no-solution?`, that processes the list of singleton clauses in the given formula is used to determine if this is the case.

If the formula contains one or more singletons, the function assigns a value to a single variable, simplifies the formula based on the assignment made, and recursively searches for assignments using the simplified formula. If the formula does not contain any singleton clauses, then it must only contain disjunctions. In this case, a backtracking strategy is used. The first clause's first singleton is assigned a value, the formula is simplified based on the assignment, and the simplified formula is solved. If this search is successful, then the found list of bindings is returned. Otherwise, the first clause's first singleton is assigned the complement of its first assignment, the formula is simplified based on this new assignment, and the simplified formula is solved. If the second search fails, then the formula is not satisfiable. If this search is successful, then the formula is satisfiable. Therefore, the result of this second search for assignments is always returned.

86.4.2 Testing

To test the 2-satisfiability solver, the following parse trees are defined:

```
(define F0 (parse-iformula '(and)))

(define F1 (parse-iformula '(and x
                                 (or (not y) (not x))
                                 (or (not z) y)
                                 y)))

(define F2 (parse-iformula '(and (or x1 x2)
                                 x1
                                 (or (not x1) (not x2))
                                 (or (not x1) x2))))

(define F3 (parse-iformula '(and (or a b)
                                 a
                                 (or (not a) (not c))
                                 (not c))))
```

```
(define F4 (parse-iformula '(and (or x1 x2)
                                 (or (not x1) (not x2))
                                 (or x2 x3)
                                 (or (not x1) (not x3)))))
```

`F0` is not satisfiable, because no variable assignments are possible. `F1` is not satisfiable, because `x` must be true. This leaves a formula that contains both `y` and its complement as clauses, which is clearly not satisfiable. `F2` is not satisfiable, because `x1` must be true, thus, simplifying to a formula that has both `x2` and its complement. `F3` is satisfiable by assigning `a` the value true and `c` the value of false. `F4` is satisfiable by assigning `x1` the value true and assigning `x2` the value true.

Based on the analysis above, tests are written as follows:

```
(check-equal? (2-satisfiability F0) 'reject)
(check-equal? (2-satisfiability F1) 'reject)
(check-equal? (2-satisfiability F2) 'reject)
(check-equal? (2-satisfiability F3) 'accept)
(check-equal? (2-satisfiability F4) 'accept)
```

86.5 The Solver Function

The solver must distinguish between four properties the given formula may have:

1. The formula is empty.
2. The formula has no solution because it contains both a variable and its complement as clauses.
3. The formula contains a singleton.
4. The formula only contains disjunctions.

If the formula is empty, then the accumulator invariant informs us that the accumulator contains the variable bindings needed to satisfy the formula. Therefore, the accumulator is returned.

To determine if the given formula has no solution, its singletons are extracted and given as input to the auxiliary predicate `has-no-solution?`. If the auxiliary predicate determines that the formula contains a variable and its complement, then the formula is clearly not satisfiable. Therefore, the empty list is returned.

To determine if a formula contains a singleton, `ormap` is used to test if any clause is a `var` or a `notvar` structure. If so, a list with the `var` structures and a list with the `notvar` structures are locally defined. Arbitrarily, the formula is simplified using the first `notvar` and solved with a new accumulator that binds the variable complemented to false. If there are only `var` structures in the formula, then it is simplified using the first `var` and solved with a new accumulator that binds the variable to true.

Fig. 106 The 2-satisfiability solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ;; formula → Boolean
  ;; Purpose: Determine if given formula has x and not x
  ;; Assumption: Given formula does not contain disjunctions
  (define (has-no-solution? a-formula) ...)

  ;; singleton → singleton     Purpose: Complement the given singleton
  (define (complement-singleton a-clause) ...)

  ;; formula singleton → formula
  ;; Purpose: Remove given singleton from given formula
  (define (simplify-formula a-formula a-clause) ...)

  ;; formula (listof (symbol Boolean) → (listof (list symbol Boolean))
  ;; Purpose: Find variable assignments to satisfy the given Boolean formula
  ;; Accumulator invariant:
  ;;  acc = (listof (variable Boolean)) with the current variable
  ;;        assignments made so far to satisfy the formula.
  (define (solve a-formula acc)
    (cond [(empty? a-formula) acc]
          [(has-no-solution? (filter (λ (c) (not (2disjunction? c)))
                                     a-formula))
           '()]
          [(ormap (λ (c) (or (var? c) (notvar? c))) a-formula)
           (let [(form-vars (filter var? a-formula))
                 (form-notvars (filter notvar? a-formula))]
             (solve (remove-duplicates
                     (simplify-formula a-formula
                                       (if (null? form-notvars)
                                           (first form-vars)
                                           (first form-notvars))))
                    (if (null? form-notvars)
                        (cons (list (var-symb (first form-vars)) #t) acc)
                        (cons (list (notvar-symb (first form-notvars)) #f)
                              acc))))]
          [else ;; a-formula only has 2disjunctions
           (let* [(fsingleton (2disjunction-s1 (first a-formula)))
                  (fvar (if (var? fsingleton)
                            (var-symb fsingleton)
                            (notvar-symb fsingleton)))
                  (not-fsingleton (complement-singleton fsingleton))
                  (sol1 (solve (simplify-formula a-formula fsingleton)
                               (cons (list fvar (var? fsingleton)) acc)))]
             (if (not (null? sol1))
                 sol1
                 (solve (simplify-formula a-formula not-fsingleton)
                        (cons (list fvar (notvar? fsingleton)) acc))))]))
  (if (not (null? (solve a-formula '()))) 'accept 'reject))
```

Fig. 107 Simplifying a 2-satisfiability formula

```
;; formula singleton → formula
;; Purpose: Simplify the given formula by removing the given singleton
;; Assumption: The given formula does not have the complement of the given
;;             singleton as a clause and the given singleton's variable is
;;             assigned a value that makes the singleton true
(define (simplify-formula a-formula a-clause)
  (define (simplify a-formula)
    (cond [(empty? a-formula) '())]
          [(equal? (first a-formula) a-clause)
           (simplify-formula (rest a-formula) a-clause)]
          [(not (2disjunction? (first a-formula)))
           (cons (first a-formula)
                 (simplify-formula (rest a-formula) a-clause))]
          [(or (equal? (2disjunction-s1 (first a-formula)) a-singleton)
               (equal? (2disjunction-s2 (first a-formula)) a-singleton))
           (simplify-formula (rest a-formula) a-singleton)]
          [(equal? (2disjunction-s1 (first a-formula))
                   (complement-singleton a-clause))
           (cons (2disjunction-s2 (first a-formula))
                 (simplify-formula (rest a-formula) a-clause))]
          [(equal? (2disjunction-s2 (first a-formula))
                   (complement-singleton a-clause))
           (cons (2disjunction-s1 (first a-formula))
                 (simplify-formula (rest a-formula) a-clause))]
          [else (cons (first a-formula)
                      (simplify-formula (rest a-formula) a-clause))]))
  (remove-duplicates (simplify a-formula)))
```

Finally, if the given formula only contains disjunctions, then the first disjunction's first singleton is used to simplify the formula. An attempt is made to solve the simplified formula by, arbitrarily, binding the variable in the first singleton to the value that makes the first singleton true. If a solution is found using the simplified formula, then it is returned. Otherwise, the first disjunction's first singleton's complement is used to simplify the formula. An attempt is made to solve the simplified formula by binding the variable in the first singleton to the value that makes the first singleton false. The result obtained from attempting to solve this simplified formula is returned. The result of the design so far is displayed in Fig. 106.

86.5.1 Simplifying a Formula

The function `simplify-formula` is used to simplify a formula. It needs as input a formula and a singleton and returns a formula. A formula is simplified by removing the given singleton and by simplifying disjunctions that contain the given singleton or that contain the given singleton's complement. Any disjunction that contains the given singleton is removed given that it is true

(recall that the variable in the given singleton is assigned a value that makes the given singleton true). Any disjunction that contains the given singleton's complement are simplified to be the other variable in the disjunction. This follows from observing that the given singleton's complement plays no role in making the disjunction true. All other clauses are left unchanged in the simplified formula.

Observe that when a formula is simplified, it may contain repeated singletons. For example, consider simplifying the formula when the given singleton is `x1`:

```
(and x1 (or (not x2) (not x1)) (not x2))
```

The assumption is that `x1` is bound to true. Therefore, the first clause may be removed. The second clause simplifies to `(not x2)` because `(not x1)` is false. Finally, the third clause is left unchanged because it does not involve `x1`. The simplified formula that looks as follows:

```
(and (not x2) (not x2))
```

Only one of the repeated clauses is needed to solve the formula, and therefore, repetitions may be safely removed to obtain the simplified formula:

```
(and (not x2))
```

The function, therefore, simplifies a formula and removes duplicates.

The function `simplify` performs the simplification using the given singleton by traversing the formula. It dispatches on seven conditions:

1. The formula is empty.
2. The formula's first clause is the given singleton.
3. The formula's first clause is not a disjunction.
4. The formula's first clause is a disjunction that contains the given singleton.
5. The formula's first clause is a disjunction whose first singleton is the given singleton's complement.
6. The formula's first clause is a disjunction whose second singleton is the given singleton's complement.
7. The formula's first clause is a disjunction that does not contain the given singleton.

If the formula is empty, there is nothing more to simplify, and the empty formula is returned. If the first clause is the given singleton, then it is not included in the simplified formula, and the rest of the formula is recursively simplified.

If the first two conditions fail and the first clause is not a disjunction, then the first clause is included in the simplified formula. It is added to the result obtained from recursively processing the rest of the formula.

If the first three conditions fail and the first clause is a disjunction that contains the given singleton, then the disjunction must be true. This means

that it may not be added to the simplified formula. The simplified formula is obtained by recursively processing the rest of the formula.

If the first four conditions fail and the first disjunction contains the complement of the given formula, then the disjunction is simplified to be the other singleton in the disjunction. This singleton is added to the result of recursively simplifying the rest of the formula.

If the first six conditions fail, then the formula's first clause must be a disjunction that does not contain the given singleton. Therefore, it is added to the result of recursively processing the rest of the formula. The result of this design is displayed in Fig. 107.

The remaining two auxiliary functions are fairly straightforward to write. The first, `has-no-solution?`, checks if a formula only containing `singleton`s is empty or not. If it is empty, then false is returned, given that the formula does not contain as clauses a variable and the variable's complement. If it is not empty, then it checks if the first singleton's complement is a member of the rest of the formula. If so, true is returned because the formula is not solvable. Otherwise, the result of recursively processing the rest of the formula is returned. The predicate is written as follows:

```
;; formula → Boolean
;; Purpose: Determine if given formula has x and not x
;; Assumption: Given formula does not contain disjunctions
(define (has-no-solution? a-formula)
  (cond [(empty? a-formula) #f]
        [(member (complement-singleton (first a-formula))
                 (rest a-formula))
         #t]
        [else (has-no-solution? (rest a-formula))]))
```

A singleton's complement is constructed by determining if the given singleton is a `var` structure. If so, a `notvar` structure with the same symbol contained in the given singleton is constructed. Otherwise, a `var` structure with the same symbol contained in the given singleton is constructed. The function is written as follows:

```
;; singleton → singleton
;; Purpose: Return the complement of the given singleton
(define (complement-singleton a-singleton)
  (if (var? a-singleton)
      (notvar (var-symb a-singleton))
      (var (notvar-symb a-singleton))))
```

86.5.2 The 2-Satisfiability Problem Is in $\mathcal{P}$

To demonstrate that the 2-satisfiability problem is in $\mathcal{P}$, it is useful to first determine the complexity of the auxiliary functions. Let us call the number

of variables in the given formula `n`. We assume that the formula is well constructed. That is, it does not contain repeated clauses nor are the singletons in a disjunction the same. Regardless of the singleton subtype received as input, `complement-singleton` performs a constant number of operations.

The predicate `has-no-solution?` receives as input a list of singletons. For each singleton, it checks if it is a member of the rest of the given list. This means that the first element of the list must be compared with each element in the list after it. In the worst case, the input list is of length `n`. This means that the first element of the given list must be compared with `n-1` elements, the second element must be compared with `n-2` elements, and so on until the last element which is compared with zero elements. The total number of comparisons, therefore, is $\Sigma_{i=1}^{n}$ `i-1`. This summation is equal to $\frac{(n-1)*n}{2}$. Thus, the number of operations performed is proportional to $\mathtt{n}^2$.

The function `simplify-formula` uses `simplify` to simplify a formula and then removes duplicates from the simplified formula. We can observe that `simplify` uses structural recursion on the given formula performing a number of operations bounded by a constant for each formula element. Therefore, the number of operations is proportional to `n`. To remove duplicates, each formula element is compared with each element after it in the formula. As with `has-no-solution?`, this means that the first element must be compared with `n-1` elements, the second element must be compared with `n-2` elements, and so on until the last element which is compared with zero elements. Thus, the number of operations performed is proportional to $\mathtt{n}^2$.

Recursive calls to `solve` require, at least, a call to `has-no-solution?` and the evaluation of and `ormap`-expression that traverses the given formula. For each formula element, a constant number of operations are performed. This means that the number of operations is proportional to $\mathtt{n}^2+\mathtt{n}$ or simply proportional to $\mathtt{n}^2$. If the given formula contains a singleton, then the formula is simplified by removing a singleton. Thus, when there is a singleton in the formula, the total number of operations is proportional to $\mathtt{n}^2$. If the given formula does not contain a singleton, then the formula is simplified at most twice by removing a singleton. Thus, when there are no singletons in the formula, the total number of operations is proportional to $2\mathtt{n}^2$ or simply $\mathtt{n}^2$. This informs us that regardless of the stanza evaluated in `cond`-expression, the number of operations is proportional to $\mathtt{n}^2$. To establish the complexity of `solve`, we must bound the number of recursive calls. Observe that for each variable in the formula, at most two recursive calls are made. For each of these calls, a variable is removed from the formula. Therefore, in the worst case, `2n` recursive calls are made. Thus, we may conclude that the number of operations is proportional to $2\mathtt{n}*\mathtt{n}^2$. That is, the number of operations is bounded by $\mathtt{n}^3$, and we may conclude that the 2-satisfiability problem is in $\mathcal{P}$.

87 A Language Not in $\mathcal{P}$

We can also ask if a language is in P: Does M accept w performing a polynomial number of steps? We can recast this problem as asking if the pair (M w) is a member of the following language:

$$\texttt{L} = \{\texttt{(M w)} \mid \texttt{M accepts w performing at most } 2^{\texttt{n}} \texttt{ steps} \wedge \texttt{n} = |\texttt{w}|\}$$

Diagonalization is used, just as done for the Halting problem in Sect. 79, to prove that $\texttt{L} \notin \mathcal{P}$.

Theorem 3 *$L \notin \mathcal{P}$.*

Proof

Assume $\texttt{L} \in \mathcal{P}$. This means that the following language is also in $\mathcal{P}$:

$$\texttt{L}_1 = \{\texttt{"M"} \mid \texttt{M accepts "M" performing at most } 2^{|\texttt{n}|} \texttt{ steps} \wedge \texttt{n} = |\texttt{"M"}|\}$$

"M" is the encoding of M that may be executed by a ctm. $\mathcal{P}$'s closure under complement (see Sect. 85.2) informs us that $\texttt{L}_1$'s complement is also in $\mathcal{P}$. This means there is a deterministic tm, $\texttt{M}^*$, which decides the following language:

$$\bar{\texttt{L}}_1 = \{\texttt{"M"} \mid \texttt{M fails to accept "M" performing at most } 2^{|\texttt{n}|} \texttt{ steps} \vee \texttt{M is not the description of a tm}\}, \texttt{where n} = |\texttt{"M"}|$$

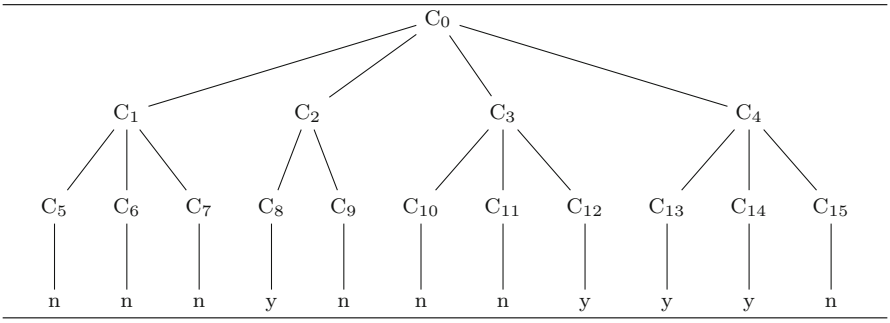
By assumption, $\texttt{L} \in \mathcal{P}$, and therefore, M operates performing a polynomial number of steps. Thus, so does $\texttt{M}^*$. Now, consider what happens when $\texttt{M}^*$ is given an encoding of itself as input. If $\texttt{M}^*$ accepts $\texttt{"M}^*\texttt{"}$, then, given that $\texttt{M}^*$ decides $\bar{\texttt{L}}_1$, $\texttt{M}^*$ fails to accept performing at most $2^{|\texttt{n}|}$ steps. This is clearly a contradiction, because $\texttt{M}^*$ accepts performing a polynomial number of steps. If $\texttt{M}^*$ rejects, then $\texttt{M}^*$ accepts $\texttt{"M}^*\texttt{"}$, performing at most $2^{|\texttt{n}|}$ steps. This is a contradiction, because $\texttt{M}^*$ only accepts if it fails to perform at most a polynomial number of steps. Whether $\texttt{M}^*$ accepts or rejects $\texttt{"M}^*\texttt{"}$, we reach a contradiction. Therefore, our assumption is wrong, and $\texttt{L} \notin \mathcal{P}$. □

3 Prove that P is closed under union.

4 Prove that P is closed under concatenation.

5 Prove that P is closed under intersection.

Fig. 108 Computation tree for a nondeterministic Turing machine



88 The Class $\mathcal{NP}$

A primary goal of complexity theory is to discover mathematical approaches to establish that solutions to practical problems are not in `P`. Problems with such solutions are, arguably, common. One such problem we have discussed that appears not to be in `P` is the traveling salesman problem (see Sect. 84). You may have studied others in an algorithms course such as integer factorization, the subgraph isomorphism problem, and even the general Boolean satisfiability problem (i.e., not the special case 2-satisfiability problem). We use the word *appears*, because the best known algorithm implemented as a deterministic Turing machine takes an exponential number of steps to find the solution, but a nondeterministic Turing machine finds the solution in a polynomial number of steps. This is what makes establishing that a problem is not in `P` hard. Separating nondeterminism and determinism in terms of polynomial running time is one of the biggest and most profound problems in computer science. It is, to date, an open research question and remains unanswered.

To explore problems not in $\mathcal{P}$, it helps to formally define what it means that a computation by a nondeterministic Turing machine is bounded by a polynomial number of steps. A nondeterministic Turing machine, `M`, is bounded by a polynomial, `p(n)`, if there is no (possible) computation that takes more than `p(n)` steps. We define the nondeterministic polynomial class of languages, `NP`, as those languages decided by a polynomially bounded nondeterministic Turing machine. In this context, by computation, we mean any series of steps that lead to either accept or reject.

Recall that a deterministic Turing machine may simulate a nondeterministic Turing machine by performing a breadth-first traversal of the computation tree for a nondeterministic Turing machine as the one in Fig. 108. In Fig. 108, there are 11 different possible computations the deterministic machine must simulate. Each node is a configuration, the edges represent a step, and `y` and

n denote, respectively, accept and reject. Nondeterministic choices are captured by more than one edge out of a configuration. Time is measured by the number of steps. In this example, the input is accepted after three steps. The figure ought to make clear why nondeterminism in Turing machines is so powerful. Only the configurations on a single path to accept need to be visited, and no configurations beyond the starting configuration need to be visited if all paths reject. In contrast, a deterministic Turing machine must systematically visit all configurations on all paths until an accepting path is found or all paths reject. Observe that a tree has an exponential number of leaves. Therefore, in the worst case, the deterministic machine performs an exponential number of steps.

89 The Boolean Satisfiability Problem Is in $\mathcal{NP}$

Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$. We shall demonstrate that it is $\mathcal{NP}$. That is, we shall describe a nondeterministic Turing machine that decides the Boolean satisfiability problem. Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem. This is equivalent to describing a single tape nondeterministic Turing machine, as you know, because a multitape machine may be simulated by a single tape machine.

To simplify the design, we assume that the input on Tape 1 is a well-formed Boolean formula in conjunctive normal form. Therefore, there is no need to check if the input represents a valid formula. We represent true as T and false as F. Finally, we use n to denote the number of (distinct) variables in the formula and m to denote the number of clauses in the formula. The machine operates in three stages:

1. For every distinct variable in the formula on tape 1, write an x on tape 2.
2. Nondeterministically, write an assignment on tape 2. Each x on tape 2 is nondeterministically substituted with a T or an F.
3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable, and the machine accepts. Otherwise, it rejects.

Stage 1 is accomplished deterministically in a polynomial number of steps. For each variable in a clause, prior clauses are checked to determine if they contain the variable. If so, an x is not written on tape 2. Otherwise, an x is written to tape 2. This phase is tantamount to determining if a variable is a member of the clauses processed so far. The number of steps is proportional to n^2. Stage 2 is accomplished nondeterministically in n steps by traversing the xs on tape 2. Stage 3 is done deterministically by processing the m clauses. For each clause, its singletons are traversed to determine if any evaluates to

true. If all clauses are processed, then the machine accepts. To process a clause, we observe that the number of singletons is bounded by `2n`. These singletons are traversed. For each, the value of its variable is plugged in. If it evaluates to true, then the machine moves to the next clause. If it evaluates to false, then it moves to the next singleton in the clause. If none of the singletons evaluate to true, then the machine moves to reject. The number of steps for stage 3, therefore, is proportional to `m*2n`. In summary, the number of operations is proportional to $\texttt{n}^2$ for stage 1 + `n` for stage 2 and $\texttt{(max n m)}^2$ for stage 3. This establishes that the nondeterministic machine performs a polynomial number of steps, and therefore, the Boolean satisfiability problem is in $\mathcal{NP}$.

90 Unsolved Problems

In general, demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps. An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement? The answer to this question is currently unknown. Contrast this with the fact that $\mathcal{P}$ is closed under complement as demonstrated in Sect. 85.2. Many properties of $\mathcal{NP}$ remain elusive to establish, and this is an area of active research.

One property that is immediately obvious, however, is that $\mathcal{P} \subseteq \mathcal{NP}$. This follows by observing that a deterministic Turing machine is a nondeterministic Turing machine whose transition relation is a function. This observation suggests another question: Is $\mathcal{P} = \mathcal{NP}$? This question remains unanswered and is one of the biggest unsolved problems in computer science. Intuitively, perhaps, we are tempted to claim nondeterminism is such a powerful feature that the answer is that they are not equal. Why would this be a tempting conclusion? A computation tree has an exponential number of paths, and a deterministic Turing machine, in the worst case, needs to simulate all these paths, thus performing an exponential number of steps. It would be truly remarkable, indeed, if a deterministic Turing machine could do the same in a polynomial number of steps. Is it even reasonable to believe this may be possible? One could theorize that a deterministic Turing machine may be able to prune the computation tree bounding the number of steps to a polynomial. No such general pruning technique is known.

6 Demonstrate that the traveling salesman problem is in $\mathcal{NP}$.

7 Prove that $\mathcal{NP}$ is closed under union.

8 Prove that $\mathcal{NP}$ is closed under concatenation.

9 Prove that $\mathcal{NP}$ is closed under Kleene star.

Part V
Epilogue

Chapter 21
Where to Go from Here

Congratulations! You have completed your first steps into the thought-provoking world of theoretical computer science. You have learned about different models of computation: their powers and their limitations. You have also learned about nondeterminism and the role it plays in computer science and in programming. The models you have studied have varied applications across computer science. Some of these applications you have seen in practice such as finding a pattern in a word (as done in Sect. 25.1) and parsing (as done in Sect. 86.3). Other applications of automata, grammars, and regular expressions include software verification, distributed systems, real-time systems, compilers, artificial intelligence, and natural language processing. This, of course, is not an exhaustive list, but it does provide a wide range of topics you are now better prepared to study. Perhaps, most importantly, you now understand how machines compute functions, how decidability problems may be solved, and the meaning of the word *algorithm*. Understanding what an algorithm is has naturally led to thinking about what can and cannot be computed and what it means for a solution to be practical. In the process, you have become a better programmer, and you have begun to explore some of humanity's limitations.

Where do you go from here? Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory. In fact, your instructor may have extended your course with modules that are not covered in this textbook. For instance, is it possible for a `cfg` to always derive a word or state that the word is not in the language? It turns out that the answer to this question is almost always yes. Any `cfg` may be transformed to Chomsky normal form and for "long" words, there is an algorithm that either returns its derivation, if it exists, or states that the word is not in the `cfg`'s language. If you have not done so already, learn about transforming a `cfg` to Chomsky normal form and how the trans-

M. T. Morazán, *Programming-Based Formal Languages and Automata Theory*, Texts in Computer Science,
https://doi.org/10.1007/978-3-031-43973-5_21

formed grammar may be used to decide if "long" words are in the grammar's language. Equally intriguing, there are many examples of undecidable problems you can explore. For example, consider a set of dominos such that each domino has two words on the same face (e.g., one of the top and one on the bottom). Ask yourself: can dominos be placed in a row (repetitions allowed) such that the appending of top strings is the same as the appending of the bottom strings? This fun puzzle is known as the post-correspondence problem, and believe it or not, it is an unsolvable problem. If you have not done so already, learn about the post-correspondence problem and other fascinating undecidable problems.

There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook. Have you ever heard of quantum finite automata? Indeed, automata theory has applications in quantum computing. Automata theory also has applications in program verification. There is a special kind of invariant known as a *preserved invariant*. A preserved invariant is one such that if it holds for some state `Q`, then it also holds for any state reachable from `Q`. This fascinating type of invariant may be used to prove that software is correct. State-based machines also have alluring applications in artificial intelligence. For example, *neural* `tm`s are used to model artificial neural networks that exhibit temporal dynamic behavior. If you do not know what the previous sentence means, then avail yourself of the opportunity to engage in the process of discovery by researching the topic. Have fun!

In closing, you are now well equipped to explore formal languages and automata theory applications in many fields of computer science as well as to deepen your understanding of computer science's theoretical underpinnings. You may do so by either taking more advanced courses or by personal inquiry. Regardless of your approach, you bring with you actual programming experience with state-based machines, grammars, and regular expressions. Mind what you have learned, and apply it in your future intellectual endeavors. Above all, enjoy the challenges that come with process of discovery and the rigor required to prove your designs correct!