A Very Simple Grammars Book

Philip W. Howard

September 24, 2019

Contents

List of Figures

Preface

I've been teaching Introduction to Grammars (aka Computer Theory) at Oregon Institute of Technology (OIT) for several years. OIT offers a very hands-on program, so why include a theory course?

There is a Computer Theory text that I really like, but it has become criminally expensive (as I'm writing this, Amazon would be glad to sell you a new paperback edition for over $300). So, why not write my own? Lot's of reasons that I won't enumerate, but obviously I've attempted to get past those and actually write the book.

This section needs work. I hope that this book proves useful to my students, and if it is discovered by others, that it is useful to them as well.

Phil Howard

Part I

Introduction

My grammars students often wonder why we include a theory course in an otherwise very hands-on program. My answer, "Because grammars is fun!" doesn't seem to carry much weight, so I offer the following list of benefits to studying this topic:

1. The grammars course lays the theoretical foundation for the compilers course. What does a compiler do? It implements several formal language processors. What is a formal language? That's part of what is learned in the grammars course.

2. You will be better at what you're being trained to do if you can think abstractly. A theory course helps you do that.

3. Many programming problems can be made much easier if you can transform the problem to a different domain. The grammars class illustrates ways this is done. If you "get" the mechanism, you can apply it elsewhere.

4. As you are stretched, you grow. By facing difficult challenges in your course work, you are better equipped to face difficult challenges in your professional work.

5. And besides, grammars is fun!

Chapter 1

Introduction

1.1 Historical Background

This section needs work.

1.2 Formal Languages

Computer theory deals with what is known as "formal languages". They aren't formal in the sense that you use them when talking to important people (as opposed to the informality that you allow when shooting the breeze with friends). They are formal in the sense that the conform to a specific (usually mathematical) form. In particular, given a statement, it is always possible to answer the question, "Is that statement a valid statement in this formal language?" English does not meet this definition. Consider all the red ink used by English teachers while grading freshmen compositions. The writers thought their statements were valid English, the but teachers disagreed.

Figure 1.1 illustrates what a formal language processor does. Any arbitrary input can be fed into the processor, and it answers the question, "Is the input a valid statement in the language?". It always returns "yes" or "no". If the language is a formal language, there can be no "maybe".

Figure 1.1: A formal language process is fed some input and it returns one of two answers: The input is a valid statement in the language or The input is not a valid statement in the language.

If you attempted to create a Language Processor for English, it should probably reject statements like, "Cup sky red Perl". But could you create a processor that accepts all valid English (including English poetry), and rejects all non-English. To get a sense of the difficulty of this challenge, I refer you to poetry (considered valid English poetry by those who get to decide such things) by ee cummings (capitalization is correct).

Those who study formal languages usually restrict themselves to very simple languages. As an example, consider the language of all strings consisting of any combination of the letters "a" and "b". This language includes strings such as "aaaa" and "abababbbb". It isn't useful for anything outside the study of formal languages, but it is an example of a formal language.

The study of formal languages is tightly coupled with mathematical sets. In particular, a formal language is a set of strings. Namely, all those strings that meet the definition of the language. This book assumes you are familiar with mathematical sets including the operations union, intersection, and subtraction. If you aren't familiar with these operations, I'm sure there's a Wikipedia page that can help you out.

Formal languages consist of the following:

1. An alphabet: a set of characters that the strings in the language are composed of. The alphabet for a language is often represented by the symbol Σ.
2. A definition of what strings are in the language.

Let's illustrate with an example.

$$\Sigma = \{ab\}$$

All strings of five letters.

Given this definition, the string "aaxbb" would be rejected because 'x' is not in the alphabet. The string "aabb" would likewise be rejected because it consists of four letters, not five. However, the strings, "aaaaa", "aabbb", and "ababa" would all be included in the language (along with many others).

Formal languages can be broken into categories based on the mechanism used to specify the language. We will initially be interested in three types of definitions, which are discussed in the following chapters.

1. Recursive definitions (not to be confused with Recursive Languages discussed in Chapter IV)
2. Regular Expressions
3. Context Free Grammars

Chapter 2

Recursive Definitions

Recursion is a useful technique in writing some computer programs. It is also a useful technique in specifying formal languages. Consider the following two definitions of the set EVEN:

1. The set of all positive integers divisible by 2
2. A recursive definition:
   (a) 2 is in EVEN
   (b) If x is in EVEN then $x + 2$ is in EVEN.

Which of these two definitions is the most useful? Probably the first one. If you wanted to prove the 96 is in EVEN using the first definition, a simple division by 2 suffices. If you wanted to prove this using the second definition, it would take a bit longer. But there are other instances where a recursive definition is quite elegant.

Consider the following non-recursive definition of arithmetic expressions:

1. $\Sigma = \{number \ + \ - \ * \ / \ (\ )\}$
2. Can't have two operators in a row

3. Must have balanced parenthesis
4. Can't have two numbers in a row
5. Can't begin or end with an operator

Is this set of rules sufficient? Do they allow every valid arithmetic expression? Do the preclude every invalid expression? Could you argue from these rules that the following is a valid arithmetic expression?

$$(2+7)/3-(((4+5)*(6)-1)*3+4)$$

How about:

$$()()(2+7)/3-(((4+5)*(6)-1)*3+4)$$

This last expression meets the definition given above, but is not a valid arithmetic expression because of the empty parenthesis. We could attempt to enhance our list of rules, but how could we know that we have a complete set?

Let's instead give a recursive definition:

1. Any number is in AE
2. If f and g are in AE, then so are:
   (a) $f+g$
   (b) $f-g$
   (c) $f*g$
   (d) f/g
   (e) (f)

How can we use this definition to prove an expression is a valid arithmetic expression. We do so by construction: invoke the various rules one at a time until the desired expression is constructed. Let's do so with the expression:

$(2 + 7)/5$

The construction is as follows:

1. 2 is in AE (Rule 1)
2. 7 is in AE (Rule 1)
3. $2 + 7$ is in AE (Points 1 and 2 and Rule 2b)
4. $(2 + 7)$ is in AE (Point 3 and Rule 2e)
5. 5 is in AE (Rule 1)
6. $(2 + 7)/5$ is in AE (Points 4 and 5 and Rule 2d)

Having constructed the desired expression using the rules, we have proven that it is a valid arithmetic expression. (More formally, we have proven that it is in the language AE as defined above).

How do we prove that an expression isn't in AE? This is a bit more complicated. Take the expression given above:

$()()(2 + 7)/3 - (((4 + 5) * (6) - 1) * 3 + 4)$

Let's generalize it to the statement:

> Valid statements in AE never have empty parenthesis.

Before proving this statement, we will prove a different one:

> Valid statements in AE never start with a close parenthesis and never end with an open parenthesis.

The proof is as follows:

1. Numbers do not contain parenthesis, so Rule 1 cannot create an AE that starts with a close parenthesis or ends with an open parenthesis.
2. If f and g do not start with a close parenthesis or end with an open parenthesis, none of the Rule 2's will create a string that starts with a close parenthesis or ends with an open parenthesis.
3. Since there is no way to construct a string that starts with a close parenthesis or ends with an open parenthesis, such a string does not exist in AE.

Back to empty parenthesis: We could prove this using a similar mechanism to what we just used. However, we will use proof by contradiction just to illustrate this mechanism. With this style of proof, you assume the opposite of what you want to prove and then reason until you encounter a contradiction. Since you got to a contradiction, your assumption must have been false. Note: as we will see later, it is important that you only make a single assumption. The contradiction means at least one of your assumptions was false. If you've made multiple assumptions, you don't know which one is false.

Let's assume that a string in AE can contain empty parenthesis. Let's define s as the shortest string that contains empty parenthesis. The string s must be constructed using one of the rules. Which one? Let's consider them one at a time:

1. Numbers do not contain parenthesis, so s cannot be constructed using Rule 1.
2. The empty string is not a valid number. Rules 2a-2e add characters to the string, and no rules subtract characters from the string. Therefore, it is not possible to have an empty string in AE.
3. Rules 2a-2d do not add parenthesis to the string so they cannot create s.
4. Rule 2e will create an empty set of parenthesis if either:
   (a) f is the empty string

(b) f begins with a close parenthesis or ends with an open parenthesis

However, we've already proven that neither of these can happen.

5. Since there is no way to construct s, are assumption must be false: A string in AE cannot contain empty parenthesis.

2.1 Constructing Recursive Definitions

We've seen several recursive definitions. What do they have in common? A recursive definition is always composed of two parts:

- Base Cases that identify some set of strings in the language. The base case for AE was "Any number is in AE".
- Recursive Rules of the form, "If x is in L, the so is $f(x)$". Rules 2a-2e for AE are the recursive rules.

When constructing recursive definitions, it is important to keep in mind what the language is. For example, the language AE was a set of strings. As a result, rules 2a-2e were adding characters to the string, they were not performing arithmetic operations. The language EVEN was a set of numbers. As a result, Rule b was performing an arithmetic operation to create a new number.

If the language is a set of strings, then the recursive rules always extend or combine strings. That is, they always make longer strings, they never remove characters from the string. In other words, you should never have a rule of the form, "If x in in L, then so is x without the trailing semicolon."

2.2 Summary

Recursive definitions provide a mechanism to construct formal languages. A recursive definition consists of base cases and recursive rules that are used

to extend the language. The mechanism we presented here isn't very mathematically rigorous, so we can't prove many properties about the languages that are generated by these definitions (a short-coming that won't be found in the next several chapters). One thing we can say about languages generated by recursive definitions is that they are always infinte. That is, they always contain an infinite number of words. This is true because there is not limit on how many times a recursive rule can be invoked, and each invocation produces a new word in the language.

Add practice problems here.

Part II

Regular Languages

Chapter 3

Regular Expressions

Many programmers are familiar with regular expressions from non-compiler contexts. Examples include using an asterisk (*) as a wildcard in a file name, or specifying patterns for the `grep` utility. Different programs use different syntax for specifying regular expressions. In this chapter a minimalist syntax for all regular expressions is presented. Most programs that interpret regular expressions enhance this syntax in various ways to make writing regular expressions easier, but the added syntax does not add extra capabilities. In this chapter, we are not interested in programs that make use of regular expressions, so we don't need the syntactic sugar that many of the add. Instead, we are interested in the formal languages that can be specified using regular expressions.

Regular expressions include the following features:

Concatenation Concatenation is gluing two strings end-to-end. For example, concatenating "`ab`" with "`bc`" yields the string "`abcd`".

Alternation Alternation means to choose exactly one from a set of alternatives. Regular expressions use either the vertical bar (`|`) or the plus sign (`+`) to mean alternation. So the expression `a | b | c` means to choose either an '`a`', a '`b`', or a '`c`'.

Grouping Parenthesis can be used for grouping operations much as they can in algebraic expressions.

Kleene Closure Kleene Closure means to take zero or more instances of a string. Kleene Closure is denoted by an asterisk (`*`). So, for example, `x*` means zero or more '`x`' characters. Kleene Closure has higher precedence than concatenation so that `ab*` means `a(b*)` not `(ab)*`.

In addition to these operations, the Λ symbol is used to represent an empty string (a string with no characters in it).

The most common enhancements to this syntax are as follows:

zero or one The question mark (`?`) indicates zero or one of an item so that `a?` means the same as (Λ `|` `a`).

one or more The plus sign (`+`) is similar to Kleene Closure, but it is one-or-more not zero-or-more so that `a+` means the same as `aa*`.

character range Square brackets (`[]`) can be used to specify a character range so that `[a-m]` means any single character in the range '`a`' through '`m`'. This could be represented longhand as '`(a | b | c | d | ...)`'.

If we want a regular expression for integer constants, we could try

```
[0-9]+
```

but this allows any number of leading zeros. A better expression would be:

```
[1-9][0-9]*
```

This fixes the leading zero problem, but it does not allow the number zero. This can be fixed as follows:

```
0 | ([1-9][0-9]*)
```

If we want to allow negative numbers, we could add an optional minus sign:

```
0 | (-?[1-9][0-9]*)
```

The following exercises can be used to practice writing regular expressions. You use the enhanced syntax or the minimal syntax for these exercises.

Exercises

1. Write a regular expression for a string containing any odd number of the letter `a`.

2. Write a regular expression for C (or Java) variable names. Valid characters include upper and lower case letters, digits, and the underscore (_).

3. Write a regular expression for a string containing any number (including zero) of a positive even number of `a`'s followed by an add number of `b`'s. The following are valid strings: `aaaabbb`, `aabaabaabbb`, `aaaaaabbbbbaabaabaab`. The following are not valid strings: `aaab`, `aaaabbbaa`, `bbbaab`.

4. For the previous question, state why each of the non-valid strings are non-valid.

5. Write a regular expression for a floating-point constant. The following rules apply:

   (a) The integer part cannot have leading zeros unless the integer part is zero.

   (b) If there is a decimal point, it must be followed by at least one digit.

   (c) The decimal part must not have trailing zeros unless the decimal part is zero.

3.1 Language Families

Each regular expression defines a language.[1] Remember that a formal language is a set of strings. Also recall that it is possible to have a set of sets, so what can we say about the set of all languages that can be defined by regular expressions? This set is known as Regular Languages. They are "regular" in the sense that they can be defined by a regular expression. Regular languages have a set of properties that they share. We will eventually explore what these properties are.

Separate from the set of regular languages, there is the language of regular expressions. We can give a recursive definition of this language (but we cannot give a regular expression for it because, as we shall see, it is not regular).

1. Every letter in Σ is in RE
2. If r_1 and r_2 are in RE, the so are:
   (a) $r_1 r_2$
   (b) $r_1 + r_2$
   (c) r_1*
   (d) (r_1)

So we now have three separate but related languages:

- The language (set of all strings) defined by a particular regular expression.
- The set of all languages definable by regular expressions.
- The set of all regular expressions.

[1] These languages are not unique; it is possible to write multiple regular expressions for the same language.

It is important to keep these three languages separate. Consider the following questions, one for each category of language:

- What is the language defined by `(a | b)*a(a | b)*`? This question is asking you to enumerate (or otherwise describe) a particular regular language.
- Is the language L regular? This is asking whether the specific language L is in the set of all languages definable by regular expressions.
- Is the statement, '`a | (b | c)+` a regular expression? This is asking "Is it a well-formed regular expression?" or "Is it in the language for regular expressions that we gave a recursive definition for?"

Any language that can be defined by a regular expression is known as a regular language. In the coming sections we want to examine properties of regular languages (properties of the entire family of regular languages, not of a particular regular language. We will do a couple of them now.

First of all, how do you prove a language is regular? The obvious solution is to write a regular expression that generates the language. One needs to be careful to make sure the regular language actually generates the correct language. For example, if one asked, "Is the language of all strings over $\Sigma = \{ab\}$ which contain at least one `a` and at least one `b` regular?" the regular expression

```
(a+b)*a(a+b)*b(a+b)*
```

would not be proof that it was. All strings generated by this regular expression contain at least one `a` and at least one `b`, but the string `ba` is part of the language, but it cannot be generated by the regular expression.

A regular expression that generates a language is adequate proof that the language is regular (provided the regular expression actually corresponds to the language). How can we prove a language isn't regular? An inability to write a regular expression for the language is not sufficient proof. The statement, "I can't write a regular expression for that language" might be

because the language isn't regular or it might be because you simply aren't creative (or smart or persistent) enough to come up with one. We will Have to wait awhile before we can come up with a proof that a language isn't regular.

Theorem 1 gives the first property we will prove regarding regular languages.

Theorem 1 *All finite languages are regular.*

The proof is quite simple. If a language is finite, every word in the language can be enumerated. The list might be very long, but it can be generated. A regular expression that corresponds to this language is simply the alternation of each word in the language:

(word_1) + (word_2) + . . . + (word_n)

3.2 Why Regular Languages?

We've identified a class of languages known as "Regular Languages" — namely, those languages that can be defined by regular expressions. Why is this class of languages interesting? From a theory point of view, they are interesting because they form what is considered the smallest class of mathematically defined formal languages. Later chapters will deal with larger classes of languages which are supersets of regular languages.

From a practical point of view, regular languages are useful because fast an efficient to process. By "process", what is meant is answering the question, "Is this input a member of a particular regular language?" A common application of this is the `grep` utility included with most Linux distributions. With `grep`, you specify a regular expression and an input file, and `grep` identifies all lines of the file that include a match for the regular expression[2]. Most uses of regular expressions include "scanning" for a match. A text string (often the

[2]This is a simplification of what `grep` does, but it is accurate for the purposes of this illustration

contents of a text file) is read looking for a match to the regular expression. How efficient are regular expressions scanners? Theorem 2, which won't be proven until later, gives a bound on their efficiency.

Theorem 2 *Any regular expression can be scanned for in $O(N)$ with small time constant where N is the length of the input string.*

The next chapter discusses a "machine" that can perform this processing.

Chapter 4

Finite Automata

There is a well-known children's game called Chutes and Ladders. The goal is to be the first player to move their piece from the beginning (bottom) to the ending (top) of the board. On a player's turn, they spin a spinner which lands on a number. The player advances their piece the specified number of squares on the board. If their piece lands on the bottom of a ladder, they climb the ladder thus making extra progress towards their goal. If their piece lands on the top of a chute, they slide down the chute winding up closer to the beginning. This game is considered a "children's" game because it involves no strategy. The outcome of the game is completely determined by the sequence of numbers generate by the spinner (assuming the players don't cheat).

What does this game have to do with regular expressions of finite automata? If we assume that the spinner only generates numbers less than ten (single digit numbers), then we could spin the spinner a bunch of times and write down each result. The result would be a string of digits. We could then ask the question, "Does this sequence of moves result in someone winning the game?" That still doesn't sound like a regular expression question (although scanning a string sounds familiar), so let's think of a state machine.

A state machine consists of a finite number of states (or configurations) and transitions from state to state. How could we turn Chutes and Ladders into a state machine? Suppose there were three players. Each player's marker could be on any of the squares on the board. Each configuration of markers

(each player on each valid square in every combination) could be considered a state. There would be a large number of states, but the number would be finite. Transitions consist of moves: for example, Player 2 moves forward three spaces. Each state would have an outbound transition for each possible spinner result for each player. We now have a state machine representation of the game, but the question still remains, "What does this have to do with regular expressions?"

Before we answer that question (through some other examples), lets define a standardized way of graphically representing state machines (or more formally, finite automata). Finite Automata (aka State Machines) consist of a finite number of states and transitions between states. One state is defined as the start state, and any number of states can be defined as final states. The start state in an FA is signified by an inbound arrow that doesn't originate from another state. Processing always starts in the start state. Final states are signified by a double circle. Transitions are labeled by the letter that is used to move from one state to another. This format is illustrated in Figure 4.1.

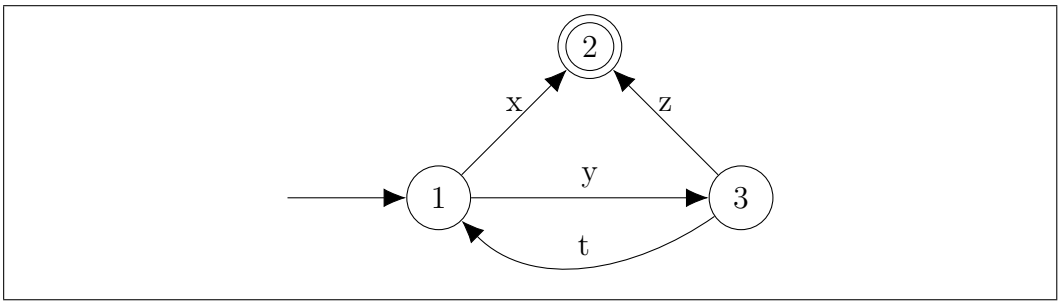


Figure 4.1: This is a sample FA. State 1 is the start state (it has an incoming arrow from nowhere). State 2 is a final state (signified by the double-circle). The transitions are labeled by what letter is used to move from one state to another.

Using the FA in Figure 4.1, and the input ytx, the processing starts in State 1 (the start state). The y is used to transition to State 3. The t is used to transition back to State 1. The x is used to transition to State 2. Since the input is exhausted while in a final state, the string is accepted by the FA.

Two conditions can cause a string to be rejected:

1. If the input is exhausted and the current state isn't a final state. The input *yty* illustrates this case.
2. If there is no outbound transition on the current letter. The input *yx* illustrates this case.

Given these rules for processing FA's, it should be clear that the FA in Figure 4.1 is equivalent to the regular expression `(yt)* (x | yz)`. It is no coincidence that there is a regular expression that corresponds to the FA. It turns out that every FA is equivalent to some regular expression, and that each regular expression has an FA that is equivalent to it. We will prove this in a later section.

Figure 4.2 gives another FA. This FA shows a slightly different format. The start state has a '-' in the label. Final states have a '+' in them. The states can be though of as follows:

1. State 1: the start state.
2. State 2: An a not preceded by an a.
3. State 3: A b not preceded by a b.
4. State 4: Have read a double letter.

Note that once a double letter has been found, the FA remains in the final state no matter what other letters are read.

FA's can also be represented in table form. These tables are called transition tables. The transition table for the FA in Figure 4.2 is given in Figure 4.3. There is one row for each state. The state name is given in the first column. The remaining columns indicate the new state based on the transition letter at the top of the column.

4.1 Non-deterministic Finite Automata

The FA illustrated in Figure 4.1 is a Deterministic Finite Automaton (DFA). It is deterministic in the sense that for each character that is processed, the

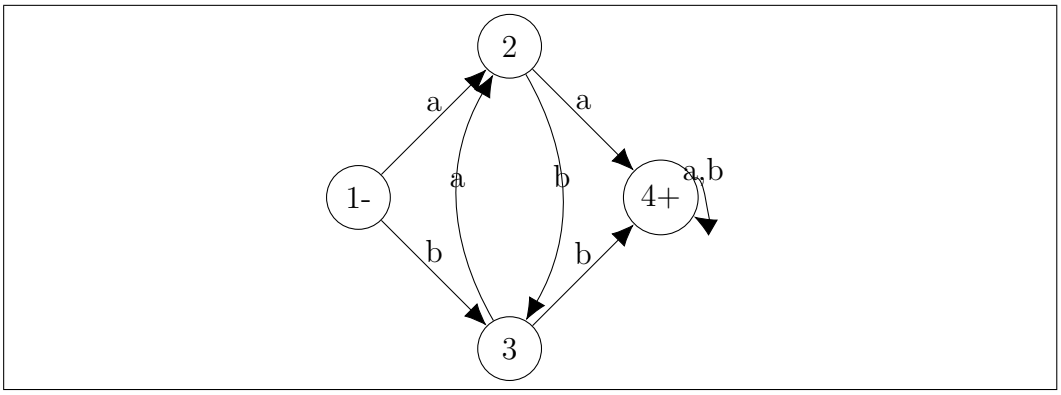


Figure 4.2: This FA accepts all strings that contain a double letter.

Current State	a	b
1-	2	3
2	4	3
3	2	4
4+	4	4

Figure 4.3: This transition table represents the same FA as Figure 4.2

FA has exactly one choice on what to do. There is another class of FA's known as Non-deterministic Finite Automata (NFA's). With NFA's, for a given input, there is the potential for multiple choices on what to do. The choices can take two forms:

1. Multiple outbound edges labeled with the same letter. If that letter is read, any of the outbound edges labeled with that letter can be taken. This is illustrated by the edges from Node a labeled z in Figure 4.4.

2. Edges labeled Λ. These edges can be taken without consuming an input character. Node a has a Λ transition meaning you can leave Node a without consuming any characters.

The NFA in Figure 4.4 accepts the following stings:

z This string is accepted by following the Λ transition to Node b and then using the z to transition to Node e.

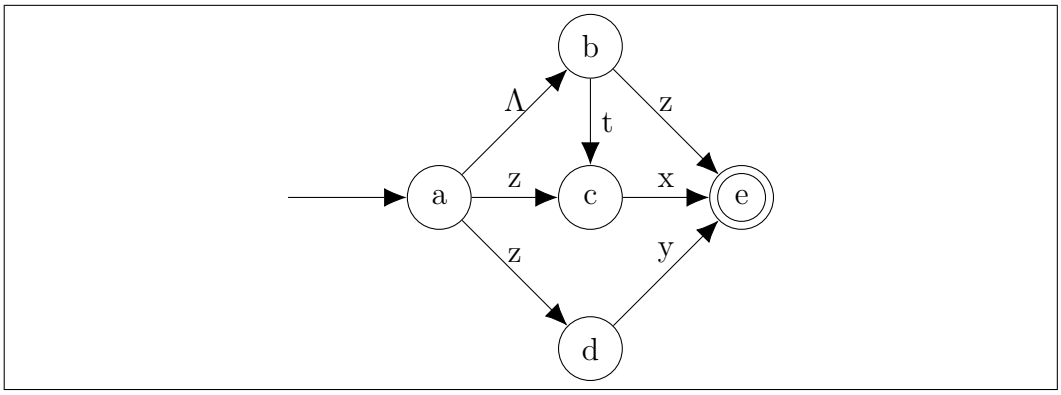


Figure 4.4: This is a sample NFA. Node a has multiple outbound transitions on z. It also has an outbound transition on Λ meaning you could leave Node a without consuming any characters.

tx This string is accepted by following the Λ transition to Node b and then using the t to transition to Node c and the x to transition to Node e.

zx This string is accepted by using the z to transition to Node c and then using the x to transition to Node e.

zy This string is accepted by using the z to transition to Node d and then using the y to transition to Node e.

NFA's aren't any more powerful than DFA's–anything you can do with an NFA you can also do with a DFA[1]. The reason for introducing NFA's is that converting from a regular expression to code that accepts strings matching that regular expression makes use of NFA's.

4.2 Thompson's

The first step in converting a regular expression to executable code is to convert it to an equivalent NFA. For this conversion, we are going to use

[1]Section Figure 4.3 gives a construction that can turn any NFA into an equivalent DFA. This is sufficient to argue that anything that can be done with an NFA can also be done with a DFA.

Thompson's Construction[2]. The beauty of Thompson's construction is that it is a mechanical process – one that doesn't require any creative thought. In other words, it can be automated. A computer program can be written to perform this conversion.

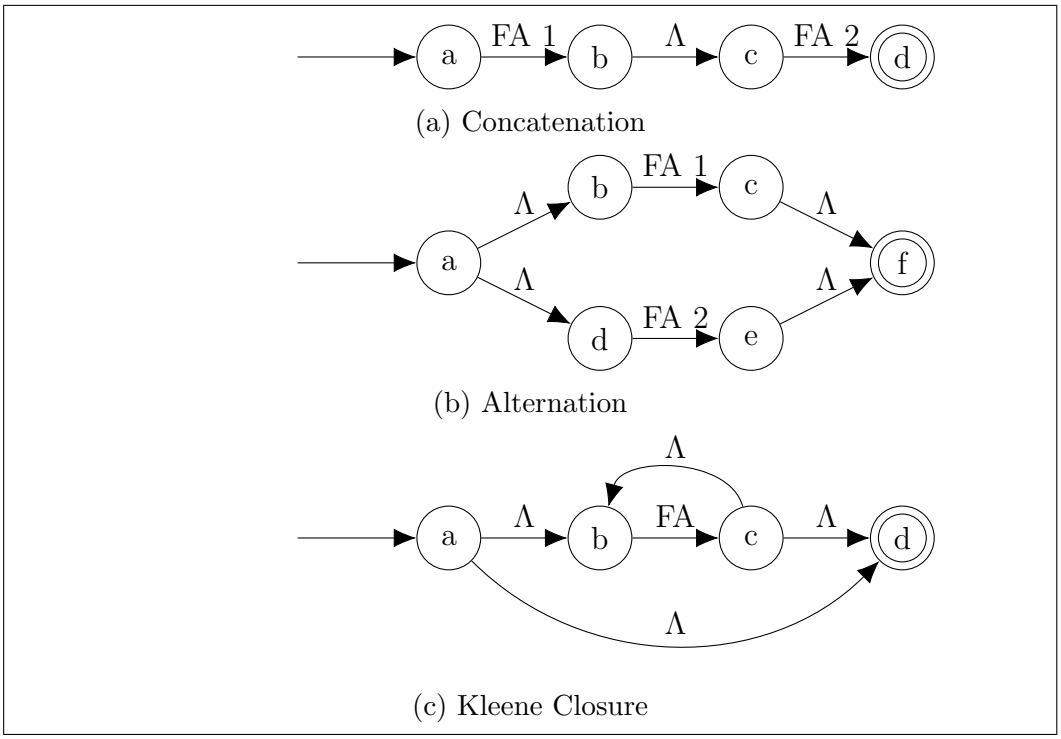


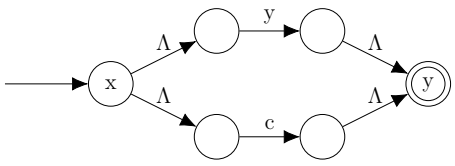
Figure 4.5: Thompson's Construction method makes use of these three diagrams. For each diagram, the FA(s) being composed are represented as two nodes (the start and end nodes) labeled FA, $FA1$, or $FA2$.

If two FA's each have a single start state and a single final state, and if the start state doesn't have any inbound edges and the final state doesn't have any outbound edges, then the two FA's can be composed by connecting the end state of one FA to the start state of the other using a Λ transition. Thompson's Construction makes use of this fact by creating composed FA's for each operation supported by regular expressions (concatenation, alternation, and Kleene Closure). An NFA can be built for any regular expression simply by composing it one small piece at a time using Thompson's three diagrams. Figure 4.5 shows the three base diagrams.

[2]Credited to Ken Thompson, the originator of the Unix operating system.

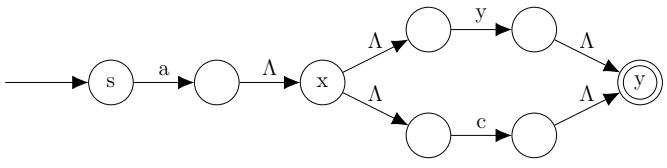
The trick to using Thompson's construction is to NOT be creative. Each FA built with Thompson's has a single start and a single final state. The diagram can be dropped as-is directly into the next step in the construction. Nodes never need to be erased, and each composition should be drawn exactly as shown in Figure 4.5.

Let's illustrate by doing several constructions. First, let's construct an NFA for `a (b | c)`. It's best to start with the inner most operation (in this case `(b | c)`) and work out. So first draw the alternation diagram as shown below:



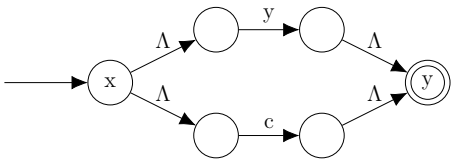
`(b | c)`

The resulting diagram gets dropped into the `FA 2` position of the concatenation diagram as illustrated below. Note how the node labeled x in the above diagram is in the location of the node labeled c in the concatenation diagram, and the y node is in the d position.



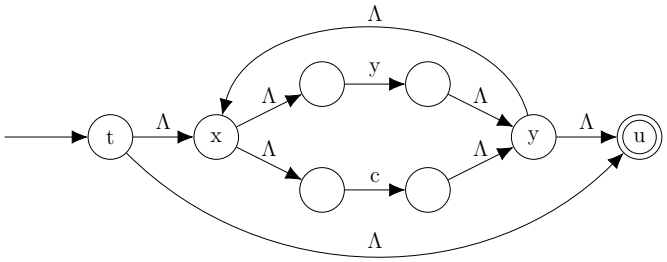
`a (b | c)`

As a second example, let's construct an NFA for `a (b | c)*`. This is the same as the previous example, except that the alternation is wrapped in Kleene Closure. The alternation is, again, the innermost operation, and it is the same as in the previous example:



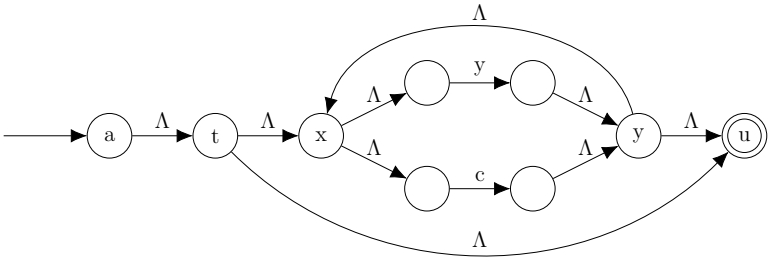
`(b | c)`

This is now dropped into the Kleene Closure construction:



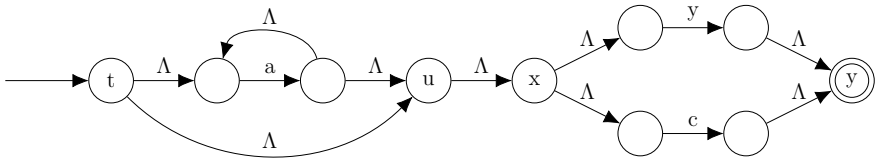
`(b | c)*`

Finally, we drop this into the concatenation construction:



`a (b | c)*`

As a third example, `a* (b | c)`, includes a concatenation of two complex item (`a*` and `(b | c)`). To construct this, first construct `a*`, then construct `(b | c)` and then use the concatenation construction to combine them. Here are the results:



a* (b | c)

It should be clear that Thompson's construction creates lots of Λ transitions. This begs the question, "isn't there an simpler way to draw these?". The answer is in two parts. If by "simpler" you mean "easier construction", my answer would be "no". The whole point of Thompson's construction is that it is a simple mechanical process. It requires no creative thought. But if by "simpler" you mean a less complex result (one without all the Λs), then the answer is "yes". The next section gives an algorithm to convert these complex NFAs into DFAs (diagrams without any Λ transitions. The goal is not to make the diagram simpler, the goal is to get a DFA because they are easier to process in code.

4.3 Subset Construction

To illustrate how an NFA could be processed, let's consider again the NFA for `a* (b | c)` presented in the last section, but presented again in Figure 4.6. In this figure, each node is labeled so they can be explicitly referred to. For each state, we can ask the question, "What states could I wind up in if I encounter a particular letter in the input?". For example, suppose we haven't consumed any input yet, what states could we be in? Clearly we could be in State 1, the start state, but because of the Λ transitions, we could also be in states $2, 4, 5, 6, 8$. This set of states ($= \{1, 2, 4, 5, 6, 8\}$)[3] forms a meta-state (let's call it A). From the meta-state A, where could we wind up if we read an a. Since the meta-state A includes state 2, we can follow the a to state 3.

[3]When specifying sets of characters, it is often easier to read the list of items if each item is separated with a comma. This works unless the set included a comma (as sets of characters for a compiler often do). I will generally included commas for readability unless the set of characters includes punctuation (commas or other punctuation marks). I hope this will improve readability.

From there we can follow Λs to $2, 4, 5, 6, 8$. This gives another meta-state. Let's call it $B = \{2, 3, 4, 5, 6, 8\}$.

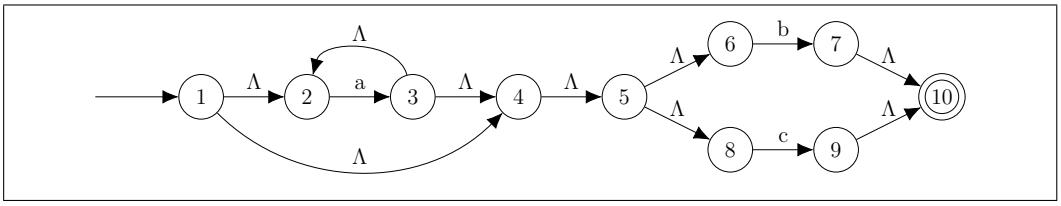


Figure 4.6: This is the NFA produced by Thompson's for the regular expression `a* (b | c)`

We could continue to find new meta-states by enumerating all possible outbound inputs from each meta-state and then following the Λs from the resulting states. The results are presented in Table 4.1.

meta-state	**NFA states**	**input**	**resulting states**
A	1, 2, 4, 5, 6, 8	a	2, 4, 5, 6, 8
A	1, 2, 4, 5, 6, 8	b	7, 10
A	1, 2, 4, 5, 6, 8	c	9, 10
B	2, 4, 5, 6, 8	a	2, 4, 5, 6, 8
B	2, 4, 5, 6, 8	b	7, 10
B	2, 4, 5, 6, 8	c	9, 10
C	7, 10	a	-
C	7, 10	b	-
C	7, 10	c	-
D	9, 10	a	-
D	9, 10	b	-
D	9, 10	c	-

Table 4.1: The results of performing the subset construction on the NFA in Figure 4.6. An input of "-" means there are no valid inputs starting from this state. Resulting states of "-" mean there are no valid destinations from this state.

We need a formal algorithm for producing these tables. The steps are as follows:

1. From the start state, follow all Λs. Follow as many as you can, not just one. This is known as the Λ closure of a state: all states reachable

from a given state following only Λs. This set of states is labeled A in the table.

2. Make multiple rows, one for each character in the source-language, for each meta-state not already in the table. Initially this is only meta-state A, and for the example NFA in Figure 4.6, the characters in the source language are a, b, c meaning three rows for each meta-state.

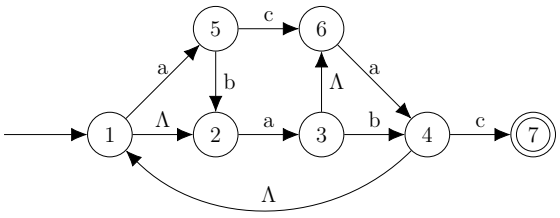
3. For each NFA state in the meta-state, if there is an outbound transition on the input, write down the destination state in the resulting states column.

4. Extend the list of states in the resulting states column by forming the Λ closure of each state already in the column.

5. If there are any new unique sets of states in the resulting states column, give them a unique label and return to Step 2.

The process will stop once all existing rows are filled in and no new rows get generated.

Let's use these rules to derive a table for the NFA below. Note: this NFA was **not** generated with Thompson's.



Step 1 yields states 1 and 2, so this set becomes meta-state-A. There are three letters in the source language (a, b, c), so this yields the following rows:

meta-state	**NFA states**	**input**	**resulting states**
A	1, 2	a	
A	1, 2	b	
A	1, 2	c	

Completing the first row, we can follow an a from state 1 to 5, and from state 2 to 3. Add these two states to the resulting states column, and then follow the Λs adding state 6. Label the set `{3, 5, 6}` as meta-state B and add rows to the table:

	meta-state	**NFA states**	**input**	**resulting states**
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	
3	A	1, 2	c	
4	B	3, 5, 6	a	
5	B	3, 5, 6	b	
6	B	3, 5, 6	c	

Completing rows 2 and 3, there are no b nor c transitions out of any of the states in meta-state A, so the resulting states for both of these rows are empty.

Moving on to row 4, we can follow an a from state 6 to state 4. We can then follow Λs from 4 to 1 and 1 to 2 yielding meta-state C containing states 1, 2, 4.

	meta-state	**NFA states**	**input**	**resulting states**
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	-
3	A	1, 2	c	-
4	B	3, 5, 6	a	C = 1, 2, 4
5	B	3, 5, 6	b	
6	B	3, 5, 6	c	
7	C	1, 2, 4	a	
8	C	1, 2, 4	b	
9	C	1, 2, 4	c	

For row 5, we can follow a b from 3 to 4 and from 5 to 2. There are no Λs from either 2 or 4, so meta-state D contains 2 and 4.

For row 6, we can follow a c from 5 to 6. There are no Λs from 6, so meta-state E contains only 6. Adding these new rows to the table, we have:

	meta-state	**NFA states**	**input**	**resulting states**
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	-
3	A	1, 2	c	-
4	B	3, 5, 6	a	C = 1, 2, 4
5	B	3, 5, 6	b	D = 2, 4
6	B	3, 5, 6	c	E = 6
7	C	1, 2, 4	a	
8	C	1, 2, 4	b	
9	C	1, 2, 4	c	
10	D	2, 4	a	
11	D	2, 4	b	
12	D	2, 4	c	
13	E	6	a	
14	E	6	b	
15	E	6	c	

Completing row 10, we can follow an a from 1 to 5 and 2 to 3. The Λ closure adds 6. This set of states is already labeled B, so we don't need to add any rows to the table.

For Row 11, there are no outbound edges on b so the resulting states is empty.

For Row 12, we can follow a c from 4 to 7. This new meta-state is labeled F.

	meta-state	**NFA states**	**input**	**resulting states**
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	-
3	A	1, 2	c	-
4	B	3, 5, 6	a	C = 1, 2, 4
5	B	3, 5, 6	b	D = 2, 4
6	B	3, 5, 6	c	E = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	F = 7
10	D	2, 4	a	
11	D	2, 4	b	
12	D	2, 4	c	
13	E	6	a	
14	E	6	b	
15	E	6	c	
16	F	7	a	
17	F	7	b	
18	F	7	c	

For row 10, we can follow the a from 2 to 3 and then the Λ to 6. This is meta-state G.

For row 11, there are no outbound transitions on b, so the resulting states is empty.

For row 12, we can follow a c from 4 to 7. This is already labeled F.

	meta-state	**NFA states**	**input**	**resulting states**
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	-
3	A	1, 2	c	-
4	B	3, 5, 6	a	C = 1, 2, 4
5	B	3, 5, 6	b	D = 2, 4
6	B	3, 5, 6	c	E = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	F = 7
10	D	2, 4	a	G = 3, 6
11	D	2, 4	b	-
12	D	2, 4	c	F
13	E	6	a	
14	E	6	b	
15	E	6	c	
16	F	7	a	
17	F	7	b	
18	F	7	c	
19	G	3, 6	a	
20	G	3, 6	b	
21	G	3, 6	c	

For Row 13, we can follow an a from 6 to 4 and then Λs to 1 and 2. This is already labeled C.

For Rows 13-18, there are no available outbound transitions, so the resulting states are empty.

The transitions for Row 19 are the same as for Row 13, so the result is C.

For Row 20, we can follow a b from 3 to 4 and then Λs give us meta-state C.

For Row 21, there are no outbound transitions on c so the resulting state is empty.

This gives us the following table:

	meta-state	NFA states	input	resulting states
1	A	1, 2	a	B = 3, 5, 6
2	A	1, 2	b	-
3	A	1, 2	c	-
4	B	3, 5, 6	a	C = 1, 2, 4
5	B	3, 5, 6	b	D = 2, 4
6	B	3, 5, 6	c	E = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	F = 7
10	D	2, 4	a	G = 3, 6
11	D	2, 4	b	-
12	D	2, 4	c	F
13	E	6	a	C
14	E	6	b	-
15	E	6	c	-
16	F	7	a	-
17	F	7	b	-
18	F	7	c	-
19	G	3, 6	a	C
20	G	3, 6	b	C
21	G	3, 6	c	-

All rows are filled in, so we are done.

Important note: The procedure is to follow the input out of the NFA states and then follow the Λs. A common mistake is to follow the Λs out of the NFA states. For example, when filling out the last row, even though there is a Λ from state 3, we don't follow it because we are only looking for transitions on c.

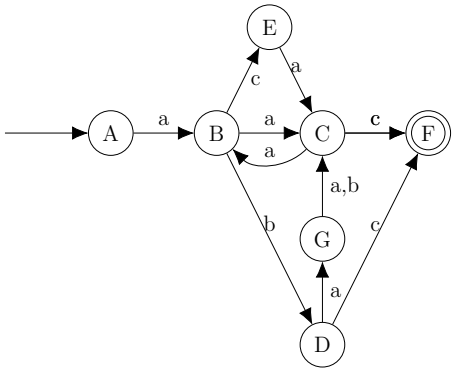
The subset construction was supposed to turn an NFA into a DFA, but it appears to have just produced a table. We can take the information in our final table from the example we just completed and summarize it in Figure 4.7.

This is known as a transition table. It is fairly straight forward to convert this table into a DFA (and a DFA into a transition table). Create states for each state in the table (the union of the From and To columns). The

From	transition on	To
A	a	B
B	a	C
B	b	D
B	c	E
C	a	B
C	c	F
D	a	G
D	c	F
E	a	C
G	a	C
G	b	C

Figure 4.7: A summary of the table derived in this section. Each row contains the start state, and input character, and the resulting state if that character is found while in the start state.

start state is always A, and any meta-state that included a final state in the original NFA is a final state in the DFA (in this case State F). The resulting DFA is as follows:



Does the subset construction always result in a DFA, or might it result in another NFA? The answer is that it always results in a DFA for the following two reasons:

1. The *input* column never has Λs, so the resulting FA will not have any Λ transitions.

2. When generating rows, each meta-state has exactly one row for each input character. As a result, there will never be multiple outbound transitions for the same character.

The consequence is that the resulting FA has no non-determinism and is therefore a DFA.

Chapter 5

Theory

What we have done so far:

We have shown that Thompson's Construction can turn any regular expression into an NFA. This implies that:

$$L(\text{regular expressions}) \subseteq L(\text{NFA})$$

This statement means that the set of all languages that can be constructed with regular expressions is a subset (or equal) of the languages that can be constructed with an NFA. In other words, anything that can be done with a regular expression can also be done with an NFA.

The Subset Construction showed that any NFA can be turned into a DFA. This implies that:

$$L(\text{NFA}) \subseteq L(\text{DFA})$$

So we now have:

$$L(\text{regular expressions}) \subseteq L(\text{NFA}) \subseteq L(\text{DFA})$$

There are proofs both by induction and construction that show that any DFA can be turned into a regular expression. The proofs aren't particularly illuminating, so they won't be presented here. The statement will simply be taken on faith. So we now have:

$$L(\text{regular expressions}) \subseteq L(\text{NFA}) \subseteq L(\text{DFA}) \subseteq L(\text{regular epxressions})$$

The only way that this can be true is if:

$$L(\text{regular expressions}) = L(\text{NFA}) = L(\text{DFA})$$

In other words, regular expressions, NFAs and DFAs are all equally powerful. They can be used to define exactly the same set of languages. This relation is known as Kleene's Theorem. It can be used to prove interesting properties of regular languages.

Theorem 3 *The union of two regular languages is regular.*

This can be proved by construction. Consider two languages L_1 and L_2 both of which are regular. Since they are regular, there is a DFA that corresponds to each of them. If we construct a new NFA by creating a new start state that is connected to the start states of both L_1 and L_2, we would have an NFA that accepts any word in L_1 or any word in L_2. Since we have an NFA for this language, the language must be regular.

This section needs work.

Theorem 4 *The intersection of two regular languages is regular.*

zzz

Part III

Context Free Grammars

Chapter 6

Context Free Grammars

This section needs work.

Chapter 7

Context Free Grammars

Regular expressions are used to define the tokens a compiler processes. We also need a mechanism to define the syntax of the language a compiler processes. Context Free Grammars are used for this purpose.

A Context Free Grammar (CFG) is made up a a list of productions of the form:

this symbol can be replaced by this collection of symbols

A production in a CFG consists of a left hand side and a right hand side. The left hand side gives the symbol that can be replaced. The right hand side gives the list of symbols that can replace the symbol on the left. The two sides are typically separated either by an arrow ($\rightarrow$) or sometimes a colon-colon-equals (::=). A sample production that indicates that the symbol `A` can be replaced by `X Y Z` is given below:

`A` $\rightarrow$ `X Y Z`

`Symbols in CFGs are of two flavors: non-terminals are those that appear on the left hand side of a production. They are non-terminals because they can be replaced by other symbols. Terminals are those symbols that never appear on the left hand side. They are ‘‘terminal’’ because they can never be replaced. For ease of reading, non-terminals are usually given in UPPERCASE. terminals are given in lowercase.`

CFGs also need a start symbol - the symbol that is the starting point for derivations. The start symbol is often either S or START, but if neither is specified, the left hand side of the first production is considered the start symbol.

1 S → a S b
2 S → λ

CFG 7.1: A CFG that defines the language of any number of a's followed by the same number of b's.

Figure 7.1 shows a complete CFG. The productions have been numbered for easy reference. There are only two productions. The first one says that the start symbol (S) can be replaced with a S b. Note that this is a recursive rule because the S appears on both sides. The second production says that S can be replaced with nothing.

What can we do with this CFG? Let's do some derivations. Starting with the start symbol and Production 1, we can get the string aSb. If we then use Production 2, we are left with the string ab. Since there are no more non-terminals, we are done.

What if we invoked Production 1 more than once? The first invocation produces aSb. The next invocation produces aaSbb. Each invocation adds another a and b. When we finally invoke Production 2, we are left with a string of a's followed by the same number of b's.

While not the most complex illustration, this CFG illustrates that CFGs are more powerful than regular expressions. Regular expressions are not able to generate balanced parenthesis. A regular expression such as (*)* allows any number of opening parenthesis and any number of closing parenthesis, but there is no way to guarantee that the number of closing parenthesis match the number of opening parenthesis. If we substituted parenthesis for the a and b in Figure 7.1, we would have a solution to the balanced parenthesis problem.

Figure 7.1 presents a more complete, and more interesting CFG. This language defines a program as zero or more statements. An individual statement can be an assignment statement (in this language, an assignment

statement is a terminal, so the assumption is that they are defined elsewhere), an if statement, or a compound statement (curly braces surrounding any number of statements).

1. PROGRAM → STMTS
2. STMTS → STMT STMTS
3. STMTS → λ
4. STMT → IF_STMT
5. STMT → COMPOUND_STMT
6. STMT → assignment_stmt
7. IF_STMT → if (expr) STMT
8. COMPOUND_STMT → { STMTS }
9. COMPOUND_STMT → STMT

Figure 7.1: This CFG defines a program as being zero or more statements, where each statement is either an `if` statement, an assignment statement or a compound statement.

```
assignment_statement
if (expr)
   assignment_statement
if (expr)
    if (expr)
       assignment_statement
if (expr)
{
   assignment_statement
   assignment_statement
   {
   }
}
```

Listing 7.2: Sample program in the language defined by the CFG in Figure 7.1

The program in Listing 7.2 illustrates the features of the language defined by the CFG in Figure 7.1. Line 1 is a simple assignment statement. The if statement that begins in Line 2 is a simple if statement. The if statement that begins in Line 4 shows a nested if statement. The compound statement that begins in Line 8 shows that compound statements can be nested and that they can be empty (in Line 11).

7.1 Derivations

Listing 7.2 claims to be a program in the language defined in Figure 7.1. How can we substantiate this claim? This is normally done by showing a derivation of the program given the CFG. Each line of a derivation substitutes a single non-terminal for the right-hand-side of a production for that non-terminal.

Derivations start with the start symbol and continue until there are only non-terminals. Each step other than the first should list the production number that was invoked to make the substitution. Rather than starting with the longer program in Listing 7.2, let's start with the shorter program in Listing 7.3.

```
1 if (expr)
  {
    assignment_statement
    assignment_statement
5 }
```

Listing 7.3: Short program program in the language defined by the CFG in Figure 7.1. The derivation of this program is given in Derivation 7.4

```
   PROGRAM
1  STMTS
4  IF_STMT
7  if ( expr ) STMT
8  if ( expr ) { STMTS }
2  if ( expr ) { STMT STMTS }
6  if ( expr ) { assignment_stmt STMTS }
2  if ( expr ) { assignment_stmt STMT STMTS }
6  if ( expr ) { assignment_stmt assignment_stmt STMTS }
3  if ( expr ) { assignment_stmt assignment_stmt }
```

Derivation 7.4: Derivation of the program in Listing 7.3

Chapter 8

Pushdown Automata

This section needs work.

Part IV

Beyond Context Free Languages

This section needs work.

Appendix A

The First Appendix

The \appendix command should be used only once. Subsequent appendices can be created using the Chapter command.

Appendix B

The Second Appendix

Some text for the second Appendix.

This text is a sample for a short bibliography. You can cite a book by making use of the command \cite{KarelRektorys}: [1]. Papers can be cited similarly: [2]. If you want multiple citations to appear in a single set of square brackets you must type all of the citation keys inside a single citation, separating each with a comma. Here is an example: [2, 3, 4].

Bibliography

[1] Rektorys, K., *Variational methods in Mathematics, Science and Engineering*, D. Reidel Publishing Company, Dordrecht-Hollanf/Boston-U.S.A., 2th edition, 1975

[2] Bertóti, E.: *On mixed variational formulation of linear elasticity using nonsymmetric stresses and displacements*, International Journal for Numerical Methods in Engineering., 42, (1997), 561-578.

[3] Szeidl, G.: *Boundary integral equations for plane problems in terms of stress functions of order one*, Journal of Computational and Applied Mechanics, 2(2), (2001), 237-261.

[4] Carlson D. E.: *On Günther's stress functions for couple stresses*, Quart. Appl. Math., 25, (1967), 139-146.

Afterword

The back matter often includes one or more of an index, an afterword, acknowledgments, a bibliography, a colophon, or any other similar item. In the back matter, chapters do not produce a chapter number, but they are entered in the table of contents. If you are not using anything in the back matter, you can delete the back matter TeX field and everything that follows it.