A Very Simple Grammars Book

Philip W. Howard

November 4, 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? The most obvious answer is that it lays the theoretical background for a compiler class. I don't find that answer particularly compelling because to be successful in a compiler class a student needs a practical knowledge of some of the concepts included in a grammars class, but the student does not need the theoretical content to be successful in compilers.

To me, the primary reason for including the grammars class in our curriculum is precisely because it is a theory class. Requiring students to think abstractly, and to work on abstract problems (some of which are quite difficult) allows them to develop skills and thought processes that can be missed in a purely practical, hands-on program of study. These additional skills and thought processes will make them better equipped to solve some of the complex problems they may face in their professional careers.

So why did I decide to write a text for this class? 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 $500). I could not, in clear conscience, ask my students to purchase a book at that price. I found an Open text that could be legally downloaded free; a print-on-demand paperback version could be purchased for under $10. It was a good text, but it was "too mathematical" for my purposes. I want my students to be able to wrestle with concepts without getting bogged down with too much notation. So I decided to attempt my own.

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 they 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".

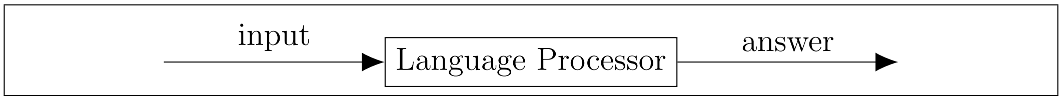


Figure 1.1: A formal language process is fed some input and it returns one of two answers: The input is a valid statement in the language or The input is not a valid statement in the language.

If you attempted to create a Language Processor for English, it should probably reject statements like, “Cup sky red Perl”. Rejecting that statement might be easy enough, 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 by ee cummings (capitalization is correct). The poetry of ee cummings is considered valid (and even good) English poetry by those who get to decide such things, but it would likely be rejected my any sensible English language validity checker.

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. Specifically, the set of all 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.

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

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 Part IV of the book)

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 they preclude every invalid expression? Could you argue from these rules that the following is (or isn't) 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 add a rule that states you can't have empty parenthesis, which would rule out the expression above. How about the expression:

$$)))4((($$

This expression has balanced parenthesis (the same number of opens as closes), but they happen to be in the wrong order. We could fix that by specifying that parenthesis have to be properly nested. Then what about the expression:

$$5(*)7$$

As you can see, there seems to be a never ending combination of ways to thwart our set of rules, so how can we know when we have a complete set? When we can't think of any new ways to break our set? What if someone more creative (or evil) comes along and finds a way to break the rules that we hadn't thought of.

Let's try a recursive definition for arithmetic expressions (called AE in the 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)

I claim (without proof) that this is a complete set of rules that always works.

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 is not 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, our assumption must be false and we can conclude that 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, then so is $f(x)$". Rules 2a-2e for AE are the recursive rules. Recursive rules can use multiple variables as illustrated in the definition of AE.

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, so 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 is in L, then so is x without the trailing semicolon.”

As another example of a recursive definition, consider the following definition of Polynomial:

1. Any number is in Polynomial
2. The variable x is in Polynomial
3. If f and g are in Polynomial, then so are:
    (a) $f + g$
    (b) $f - g$
    (c) fg
    (d) f/g
    (e) (f)

Note 1: In Rule 3c, fg is used to represent the multiplication of f and g. Polynomials are mathematical expressions, so we are using the mathematical notation for multiplication. If we were using a programming language notation, this rule would probably read $f * g$.

Note 2: like AE, this is a set of strings so the arithmetic operators in Rule 3 are simply characters, no arithmetic operations are performed.

Given this definition, it should be easy enough to show that (24/4/2) is in Polynomial (note that it is also in AE). What is the meaning of this expression? is it 3 or 12 (based on which operation is performed first). It turns out this is an erroneous question. Both Polynomial and AE are sets of strings. We can answer the question, “Is this string in the language?”, but the question “What does this string mean?” is not relevant to these languages.

In programming languages, there is syntax and semantics. One way of thinking of these two is that syntax deals with the question, “Is this a valid statement in the formal language?” Semantics deals with the question, “What is the meaning of this statement?" or even, "Does this statement have a valid meaning in the language?". At this point, we are only concerned with syntax, not semantics.

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 infinite. That is, they always contain an infinite number of words. This is true because there is no limit on how many times a recursive rule can be invoked, and each invocation produces a new word in the language.

2.3 Problems

1. Given the definition of Polynomial given in this chapter argue either that the statement $x^3 + 4x^2 - 7$ either is or is not in Polynomial.

2. The definition of Polynomial given in this chapter only allows a single variable. Extend the definition to allow two variables.

3. The language PALINDROME is the set of all strings that read the same forwards and backwards. Give a recursive definition of even-length PALINDROMEs over $\Sigma = \{a\ b\}$.

4. Extend your definition of PALINDROME to include both even and odd length strings.

5. Give a recursive definition of positive integers (or argue that it can't be done).

6. Give a recursive definition of positive rational numbers (or argue that it can't be done).

7. Give a recursive definition of positive real numbers (or argue that it can't be done).

8. Give a recursive definition for the set of strings over $\Sigma = \{0123456789\}$ that cannot start with the digit zero.

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 them 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 + ...)`'.

These enhancements will be used in a few examples, and they are allowed in homework problems provided their use is clear. Note that the one-or-more enhancement uses the plus sign that can also be used for concatenation so any regular expression that uses **any** of these enhancements must use the vertical bar for alternation.

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]*)
```

There are exercises at the end of the chapter that can be used to practice writing regular expressions. You use the enhanced syntax or the minimal syntax for these exercises.

3.1 Manipulating Regular Expressions

Regular expressions look similar to algebraic expressions, so it might be tempting to manipulate them as if they were algebraic expressions. Some manipulations are possible, but the rules can be quite different from algebra. For example, in both regular expressions and algebra,

$$a(b + c) = ab + ac$$

but the following have no algebraic equivalent:

$$(a^*)^* = a^*$$
$$(a + b^*)^* = (a + b)*$$
$$(a^*b^*)^* = (a + b)^*$$

3.2 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

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

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.

We can also think of the set of all regular expressions. This is yet another language (set of strings). 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, then 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.

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?"

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 L of all strings over $\Sigma = \{a\ b\}$ 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`. However, the string `ba` is part of L, but it cannot be generated by the specified regular expression. In other words:

$$(a+b)^*a(a+b)^*b(a+b)^* \subset L$$

This is not sufficient proof that L is regular. It is always possible to give a regular subset of any language. This can be done by giving a alternation of a finite list of words from the language. The ability to show there is a regular subset of the language is not sufficient proof that the language is regular.

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. Chapter 5 provides a mechanism to prove 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:

$$\text{(word\_1)} + \text{(word\_2)} + \ldots + \text{(word\_n)}$$

Before proving other properties of regular languages, we have a question to answer.

3.3 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 the smallest class of languages in what's known as the Chompsky Hierarchy of languages. Later parts of the text will deal with the larger classes of languages in the hierarchy, all of which are supersets of regular languages.

From a practical point of view, regular languages are useful because they are fast and efficient to process. By "process", we mean 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 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.*

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

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

3.4 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 a positive even number of `a`'s followed by an odd 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 not 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.

   For the following problems, give an English definition of the language defined by the specified regular expressions. Don't simply transliterate the regular expressions. For example, for the regular expression $a^*b(a+b)^*$, you should not say, "Any number of a's followed by a b followed by any combination of characters." Instead, you should say "All strings with a b.

   Assume $\Sigma = \{a\ b\}$ unless otherwise specified.

6. $(a+b)^*a(a+b)^*b(a+b)^*$

7. $(aa + ab + ba + bb)^*$

8. $(a + b)(aa + ab + ba + bb)^*$

9. $((a + b)(a + b))^*$

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 or 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 Finite Automaton[1] (aka state machine).

A finite automaton 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

[1] "Automaton is the singular of the plural "Automata".

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 such as “Player 2 moves forward three spaces”. Each state would have an outbound transition for each possible spinner result for each player. The finite automaton representation of the game makes it clear that there is no strategy involved. The machine process the input (the sequence of spins) without any human involvement. The outcome is determined solely by the sequence of spins. 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 finite automata. Finite automata 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.

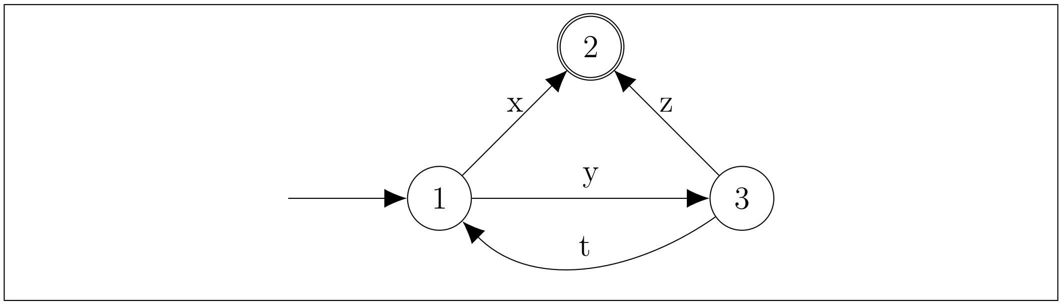


Figure 4.1: This is a sample FA. State 1 is the start state (it has an incoming arrow from nowhere). State 2 is a final state (signified by the double-circle). The transitions are labeled by what letter is used to move from one state to another.

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 can be shown 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 thought 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.

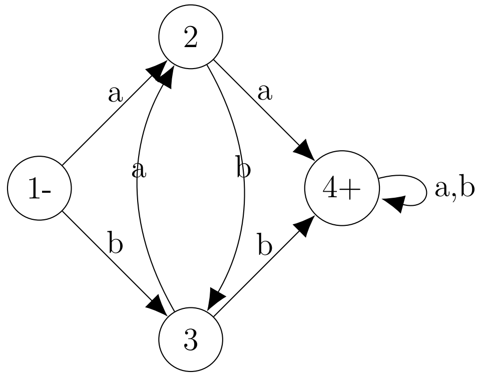


Figure 4.2: This FA accepts all strings that contain a double letter.

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

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 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:

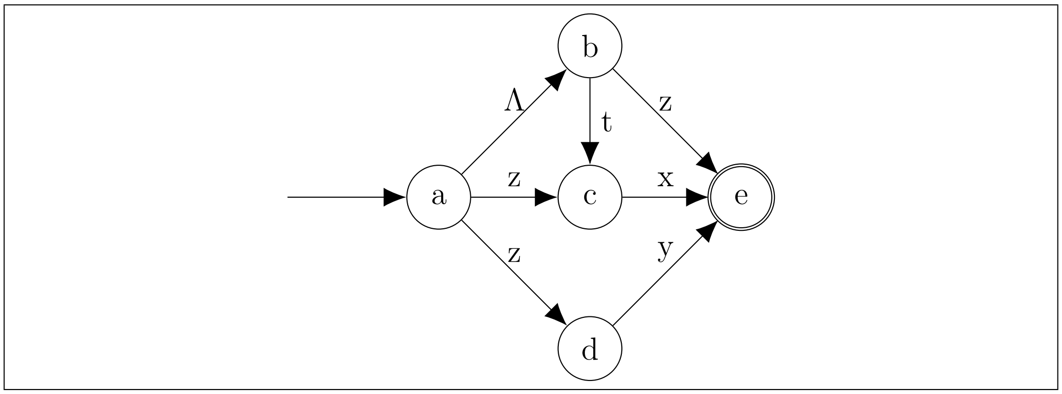


Figure 4.4: This is a sample NFA. Node a has multiple outbound transitions on z. It also has an outbound transition on Λ meaning you could leave Node a without consuming any characters.

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

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[2]. 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.

[2] Section 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.

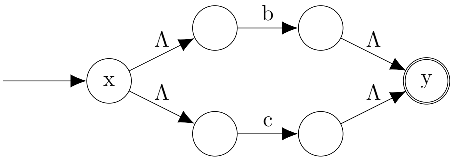
4.2 Thompson's

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 Thompson's Construction[3]. 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.

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 showing how to compose 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.

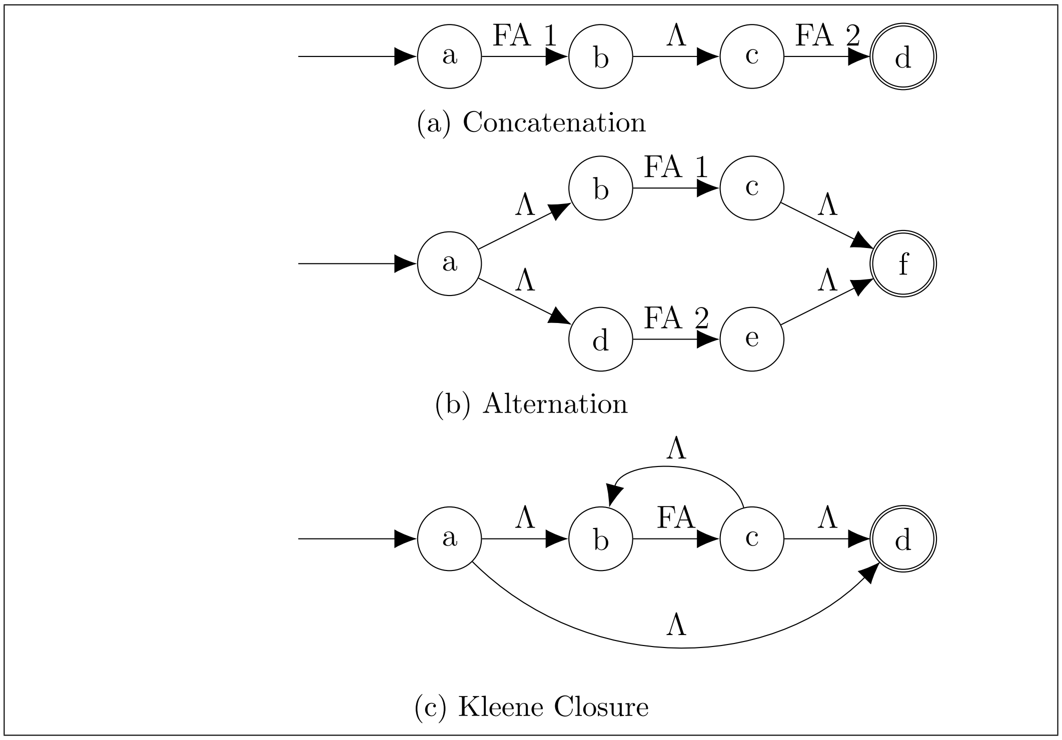
The trick to using Thompson's construction is to NOT be creative. Each FA built with Thompson's has a single start and a single final state. The diagram can be dropped as-is directly into the next step in the construction. Nodes never need to be erased, and each composition should be drawn exactly as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

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



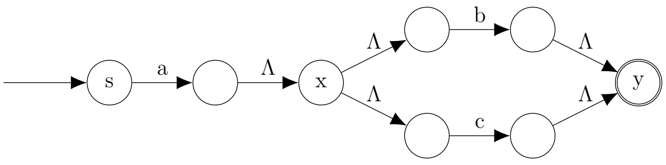
(a) Concatenation

(b) Alternation

(c) Kleene Closure

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 $FA2$.

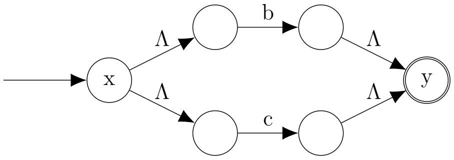
The resulting diagram gets dropped into the `FA 2` position of the concatenation diagram as illustrated below. Note how the node labeled x in the above diagram is in the location of the node labeled c in the concatenation diagram, and the y node is in the d position.



`a (b + c)`

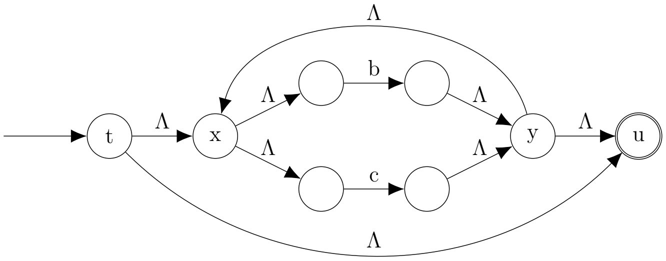
As a second example, let's construct an NFA for $a(b+c)^*$. This is the same as the previous example, except that the alternation is wrapped in Kleene

Closure. The alternation is, again, the innermost operation, and it is the same as in the previous example:



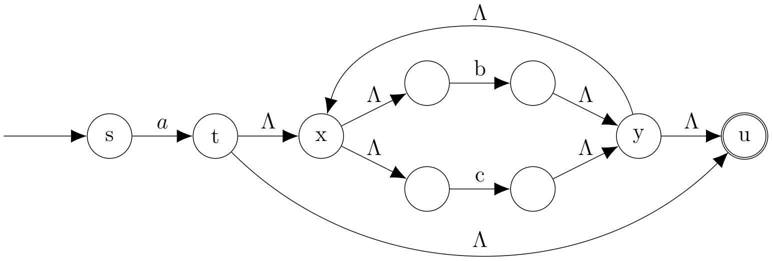
(b + c)

This is now dropped into the Kleene Closure construction:



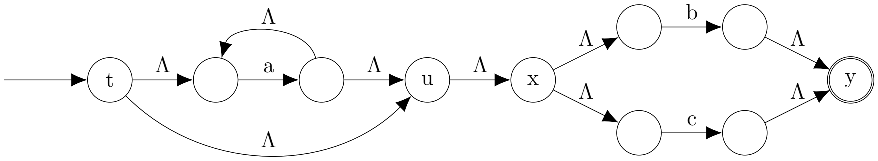
$(b+c)^*$

Finally, we drop this into the concatenation construction:



$a(b+c)^*$

As a third example, $a^*(b+c)$, includes a concatenation of two complex items a^* and $(b+c)$). To construct this, first construct a^*, then construct $(b+c)$ and then use the concatenation construction to combine them. Here is the resulting diagram:



$a^*(b+c)$

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 ($= \{1, 2, 4, 5, 6, 8\}$)[4] 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. From

[4] 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.

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

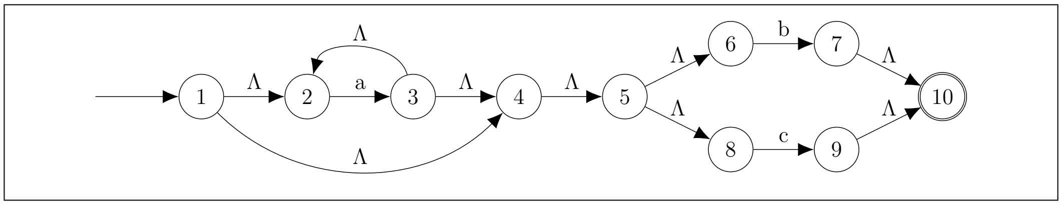


Figure 4.6: This is the NFA produced by Thompson's for the regular expression $a^*(b + c)$

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:

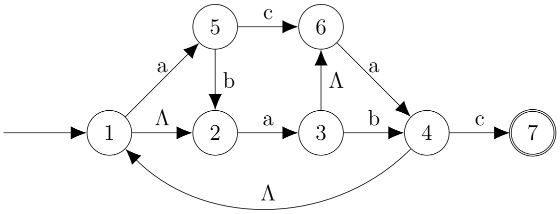
1. From the start state, follow all Λs. Follow as many as you can, not just one. This is known as the Λ closure of a state: all states reachable

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.

For row 5, we can follow a b from 3 to 4 and from 5 to 2. We can then follow a Λs from 4 to 1 and from there to 2 (which we've already reached), so this meta-state contains 1, 2 and 4, which we've already labeled C.

For row 6, we can follow a c from 5 to 6. There are no Λs from 6, so meta-state D 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	C
6	B	3, 5, 6	c	D = 6
7	C	1, 2, 4	a	
8	C	1, 2, 4	b	
9	C	1, 2, 4	c	
10	D	6	a	
11	D	6	b	
12	D	6	c	

Completing row 7, 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 8, there are no outbound edges on *b* so the resulting states is empty.

For Row 9, we can follow a *c* from 4 to 7. This new meta-state is labeled *E*.

	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	C
6	B	3, 5, 6	c	D = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	E = 7
10	D	6	a	
11	D	6	b	
12	D	6	c	
13	E	7	a	
14	E	7	b	
15	E	7	c	

For row 10, we can follow the a from 6 to 4 and then the Λ to 1 and 2. This is already labeled *C*.

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

For row 12, there are no outbound transitions on *c*, so the resulting states is empty.

	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	C
6	B	3, 5, 6	c	D = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	E = 7
10	D	6	a	C
11	D	6	b	-
12	D	6	c	-
13	E	7	a	
14	E	7	b	
15	E	7	c	

For Rows 13-15, there are no outbound edges from 7, so the resulting states are 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	C
6	B	3, 5, 6	c	D = 6
7	C	1, 2, 4	a	B
8	C	1, 2, 4	b	-
9	C	1, 2, 4	c	E = 7
10	D	6	a	C
11	D	6	b	-
12	D	6	c	-
13	E	7	a	-
14	E	7	b	-
15	E	7	c	-

All rows are filled in, so we are done.

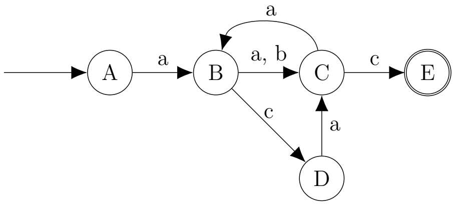
Important note: The procedure is to follow the input out of the NFA states and then follow the Λs. A common mistake is to follow the Λs out of the NFA states. For example, when filling out the last row, even though there is a Λ from state 3, we don't follow it because we are only looking for transitions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

From	**transition on**	**To**
A	a	B
B	a	C
B	b	C
B	c	D
C	a	B
C	c	E
D	a	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.

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 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. Some properties are easier to prove using DFA's, some are easier to prove using regular expressions, but since these mechanisms are equivalent (can be used to define the same set of languages), then proving a property using one mechanism means the property applies to all languages in the family.

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. Figure 5.1 illustrates this in picture form.

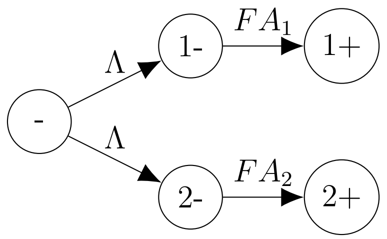


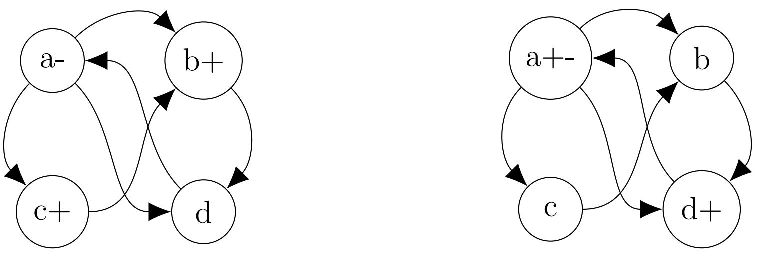
Figure 5.1: This FA accepts the strings accepted by $FA_1 \cup FA_2$

This could also be proved using regular expressions. If RE_1 is a regular expression that defines L_1, and RE_2 is a regular expression that defines L_2, then $RE_1 + RE_2$ defines the language $FA_1 \cup FA_2$.

Theorem 4 *The complement of any regular language is regular.*

First, what do we mean by the complement of a language. If a regular language L is defined over the alphabet Σ, then the complement of L, which we will call L', is the set of all strings that are in Σ^*, but not in L.

We can again prove this by construction. If L is regular, then there is a complete DFA that generates the language. By "complete", we mean that every state has an outbound transition for every letter in Σ. If we take that DFA and reverse the "plusness" of each state (that is, every state that was a final state in the original becomes a non-final state in the result, and any state that was non-final in the original becomes final in the result) then any string accepted by the original will be rejected in the final and any string regected by the original will be accepted in the final. Since the DFA is complete, it will never "crash" - that is, for each input letter, it will always have a transition it can follow. Since it is a deterministic FA, a given input can only wind up in a single state. That one an only state has opposite implications in the two FAs. If it accepts the string in one, it rejects the string in the other. As a result, the two FAs represent languages that are complements of each other.



(a) Original FA

(b) Compliment FA

Figure 5.2: A complete DFA can be modified to generate the complement of the original language by changing the "plussness" of each node: Final nodes in the original become non-final in the converted diagram and non-final nodes in the original become final nodes in the converted diagram.

Theorem 5 *The intersection of two regular languages is regular.*

DeMorgan's Law for sets states that $L_1 \cap L_2 = (L_1' \cup L_2')'$. In other words, the intersection of two sets is equal to the complement of the union of the complement of the two sets. If L_1 and L_2 are regular then so are their complements (L_1' and L_2'). Since these are regular, then so is their union ($L_1' \cup L_2'$). And finally, so is the complement of that language ($(L_1' \cup L_2')'$). Since this is equivalent of the intersection of the two languages, then the intersection must also be regular.

Theorem 6 *There is an algorithm to determine if two regular expressions are equivalent.*

Consider $L_1 = L(RE_1)$ and $L_2 = L(RE_2)$. If RE_1 and RE_2 define different languages, then either L_1 contains words not in L_2 or L_2 contains words not in L_1 (or both). Since L_1 and L_2 are regular, we can compute $L_3 = (L_1 \cap L_2') \cup (L_1' \cap L_2)$. The first term in L_3 is all strings that are in L_1 but not in L_2. The second term is all strings that are in L_2 but not in L_1. If the union of those two terms is empty, then the two languages L_1 and L_2 must be equivalent.

But now we need a decision procedure for "Is the language empty?". If we have a regular expression for a language, then the following procedure will give one word in the language:

1. Delete all stars
2. If a Λ is part of an alternation, remove that part of the alternation.
3. Delete all remaining Λ's
4. For every alternation, keep only the left-most option
5. Remove parenthesis

What is left must be a word in the language.

If we have an FA for the language, then the following procedure will determine if the FA accepts at least one word:

1. Add the start state to a work list
2. Remove a state from the work list and mark it as visited. Add all unmarked states reachable from this state to the work list.

3. Repeat Step 2 until the work list is empty.

4. If a final state was marked, then the language includes at least one word.

Theorem 7 *Let F be an FA with N states. If F accepts any words at all, then it must accept a word of length less than N.*

The shortest path from the start state to an accept state must visit each node at most once. As a result, if there is any path from the start state to an accept state, there must be a path with length less than N.

Theorem 7 leads to another decision process for “Is the language empty?”: Enumerate all words of length less than N (the number of states in the FA), and try them all. If the FA accepts any words, it must accept one of these words.

Theorem 8 *Any infinite regular language must have a loop in the FA.*

By “loop in the FA”, what we mean is that there is some set of states which can be repeated an arbitrary number of times. For example, the states 2,3,4 in Figure 5.3 form a loop that can be repeated any number of times. Consider any infinite regular language L. The language L can be generated by some DFA. The DFA is finite, so let’s say it has N states, where N is some finite number. The longest string that can be generated by this DFA without visiting the same state more than once is $\leq (N-1)$. Clearly the length can’t be $\geq N$ because if you make N transitions between N states, then at least one of those states must be repeated. The maximum length might be $< (N-1)$ because there might not be any path that touches each state exactly once.

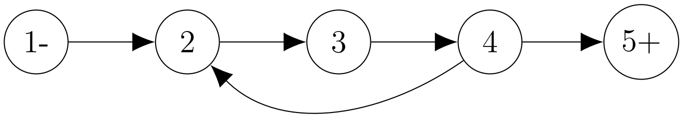


Figure 5.3: This FA represents an infinite language, and it contains a loop composed of states 2,3,4.

Since the maximum length without repeating states is $\leq (N-1)$, then any string longer than this must visit the same state at least twice. In other words, there must be a path from State S_i back to State S_i and from there to a final state. This path constitutes a loop. As a result, the FA for any infinite regular language must contain a loop.

Lemma 9 *Let L be any infinite regular language, then there exist strings x, y, z, where y is not the empty string, such that all strings $xy^n z$ with $n > 0$ are in L.*

Since there are an infinte number of words in L, any FA that generates L must have a loop. The string x represents the path from the start state to the beginning of the loop. The string y represents the path around the loop. The string z represents the path from the end of the loop to a final state. Since y represents a loop, then each circuit around the loop generates another y. Since the loop can be traversed an arbitrary number of times, strings with an arbitrary number of y's can be generated. This is shown in Figure 5.4.

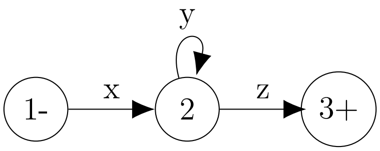


Figure 5.4: This FA illustrates the proof of the pumping lemma. The strings x, y, z are labeled. Clearly each circuit of the loop generates another y.

Lemma 9 is known as the Pumping Lemma for Regular Languages. It can be used to prove that a language is NOT regular. Before giving some examples, it is important to make several clarifications:

1. The Lemma states that all strings $xy^n z$ are in L. It does NOT say that all strings in L are in the form $xy^n z$.

2. The Lemma can be used to prove that a language is NOT regular. It cannot be used to prove that a language IS regular. In other words, there are some non-regular languages the fit the form of the pumping lemma, but any language that is infinite that does NOT fit the pumping lemma must not be regular.

Consider the language ab^*c. This is an infinite language and it is regular, so it must conform to the pumping lemma. To show this, take $x = a$, $y = b$, $z = c$. With this assignment, it should be clear that all strings xy^nz are in the language.

Now let's consider the language of all strings that consist of any number of a's followed by any number of b's. This language can be represented as a^nb^m. Is this language regular? Let's see if we can prove it is NOT regular using the pumping lemma. Let's take $x = a$, $y = ab$, $z = b$. The string xyz is $aabb$ which is in the language. However, the string xy^2z is $aababb$ which is NOT in the language. Can we conclude from this that the language is not regular? No.

The previous paragraph gave a single example of a choice for x, y, z, and a single example is not a proof. To use the pumping lemma to prove a language is not regular, you must show that there is no possible choice of x, y, z where the pumping lemma applies. Let's consider another choice. Let's choose $x = a$, $y = a$, $z = b$. With this choice, any xy^nz can be written $a^{n+1}b$, all of which are in the language. Note that this does not represent ALL words in the language, but all words of this form ARE in the language.

Since we found strings x, y, z for which all xy^nz are in the language, we failed to prove that this language is not regular. This should come as no surprise since the since the regular expression a^*b^* generates this language. Since there is a regular expression that generates the language, the language must be regular.

Now let's make a minor modification to our language. Instead of any number of a's followed by any number of b's, let's require the a's to be followed by the same number of b's. This language can be expressed a^nb^n. Note that this time both letters have the same superscript meaning there must be the same number of each. Can we find an x, y, z such that all strings xy^nz are in the language.

> Possibility 1: y consists only of a's. In this case, increasing n in xy^nz would increase the number of a's without increasing the number of b's thus generating words not in the language.
>
> Possibility 2: y consists only of b's. In this case, increasing n in xy^nz would increase the number of b's without increasing the number of a's thus generating words not in the language.

> Possibility 3: y consists of both a's and b's. In this case, increasing n in xy^nz would increase the number of a-b transitions. Since the language only allows one $a - b$ transition, this would generate words not in the language.
>
> Since y cannot be the empty string, we've enumerated all possible choices for y and none of them meet the requirements of the pumping lemma. We can therefore conclude that a^nb^n is not regular.

This language is of more than passing concern. Consider the substitution where instead of a we have an open parenthesis and instead of b we have a closing parenthesis. The language is now $(^n)^n$, which is the set of nested parenthesis. The ability to have an arbitrary number of nested parenthesis (or curly braces) is an artifact of many programming languages. The fact that you cannot generate this language with a regular expression means that the syntax of most programming languages is not regular. It also means that the syntax for regular expressions is not itself regular.

If we assume that the syntax for programming languages is formal (and it would be difficult to write a compiler if it was not: how could you decide if a source file was a valid program if the language was not formal?), then we need another class of languages to express the syntax of programming languages. This other set of languages is the subject of the next part of this book.

Add practice problems here.

Part III

Context Free Grammars

Chapter 6

Context Free Grammars

With English, there is a progression of language elements:

Letters → Words → Sentences → Paragraphs

With computers, there is a similar progression:

Characters → Tokens → Programming Languages → Algorithms

Regular expressions are suitable for defining tokens (such as integer constants or identifiers), but they can't be used for defining most programming languages because most programming languages have nesting features (parenthesis, curly braces, etc.), and regular languages are incapable of defining such features.

In this part of the book we examine another category of languages: Context Free Languages. These languages are capable of expressing nesting, and they are suitable for defining most programming languages.

6.1 Introduction to CFGs

Recall that nested parenthesis cannot be represented as a regular expression. Another example that fits the same pattern as nested parenthesis is shown in Listing 6.1

```
if (x)
{
   if (x)
   {
      if (x)
      {
         etc.
      }
   }
}
```

Listing 6.1: Nested if statements

Each of these nesting problems boils down to the language $a^n b^n$, which we've shown cannot be represented with a regular expression. This pattern can, however, be represented with a recursive definition:

1. Λ is in BAL
2. if x is in BAL, then so is axb

This suggests that languages defined recursively are more powerful (in the sense that they can define more languages) than regular expressions. We want to formalize the structure of recursive definitions so that we can prove properties of these languages much as we did for regular languages in Chapter 5. Context Free Grammars are one such formalization.

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

> X can be replaced by A B C

Where the X, A, B, C can be thought of as variables. The potential for recursion comes about because the variable being replaced (X in this example) can appear in the replacement:

> X can be replaced by A X C

More formally, 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 that symbol. The two sides are typically separated either by an arrow (→) 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 → 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`, and 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 assumed to be the start symbol.

```
1 S → a S b
2 S → Λ
```

CFG 6.2: A CFG that defines the language of any number of `a`'s followed by the same number of `b`'s.

Figure 6.2 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[1]. 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 6.2, we would have a solution to the balanced parenthesis problem.

Figure 6.3 presents a 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 }
```

CFG 6.3: 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.

The program in Listing 6.4 illustrates the features of the language defined by the CFG in CFG 6.3. 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

[1] Technically, we've only shown that CFGs can solve one problem that regular expressions can't. It remains to be shown that everything you can do with a regular expression you can also do with a CFG. Once we show that, we can conclude that CFGs are strictly more powerful than regular expressions.

that begins in Line 8 shows that compound statements can be nested and that they can be empty (in Line 11).

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

Listing 6.4: Sample program in the language defined by the CFG in CFG 6.3

6.2 Derivations

Listing 6.4 claims to be a program in the language defined in Figure 6.3. 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 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 6.4, let's start with the shorter program in Listing 6.5.

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

Listing 6.5: Short program program in the language defined by the CFG in CFG 6.3. The derivation of this program is given in Derivation 6.6

```
    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 6.6: Derivation of the program in Listing 6.5

The derivation in Derivation 6.6 is what's known as a left-most derivation because each time there were multiple non-terminals, the left-most one was replaced. It is also possible to do right-most derivations, and you can also do neither: sometimes pick the left-most non-terminal, sometimes the right-most, and sometimes one in the middle.

6.2.1 Derivation forms

The left-most nature of Derivation 6.6 wasn't that obvious because there were never more than two non-terminals in any line of the derivation. For a better example, consider the the CFG for expressions in postfix notation given in CFG 6.7. Using that grammar, Derivation 6.8 shows a derivation of the expression

$3\ 4 + 5\ 6$ - *

Derivation 6.8 is both top-down and left-most. It is top-down because it begins with the start symbol and progresses to all terminals. When a derivation is represented graphically (see Section 6.2.2), this derivation starts at the top of the diagram and progresses to the bottom. This derivation is also left-most because in each step, the left-most non-terminal is the one that was replaced.

Derivation 6.9 shows a bottom-up right-most derivation of the same expression. It is bottom up because it begins with all terminals and combines them

```
1 EXPR → POST
2 POST → POST POST OP
3 POST → number
4 OP → + | - | * | /
```

CFG 6.7: This CFG defines the syntax for postfix expressions. In production 4, the vertical bars represent alternation, so an OP can be one of the four arithmetic operators.

```
    EXPR
1   POST
2   POST POST OP
2   POST POST OP POST OP
3   3 POST OP POST OP
3   3 4 OP POST OP
4   3 4 + POST OP
2   3 4 + POST POST OP OP
3   3 4 + 5 POST OP OP
3   3 4 + 5 6 OP OP
4   3 4 + 5 6 - OP
4   3 4 + 5 6 - *
```

Derivation 6.8: Derivation of the expression 3 4 + 5 6 - *

using productions until the terminals have been reduced to the start symbol. Note that if these steps were reversed, the derivation would be top-down. In this sense, top-down and bottom-up derivations are inverses of each other. Note also that if this derivation was inverted to its top-down form, it would be a right-most derivation. In each step it is the right-most non-terminal that gets replaced. The left-most or right-most nature of a derivation is always from the top-down perspective.

```
    3 4 + 5 6 - *
3   POST 4 + 5 6 - *
3   POST POST + 5 6 - *
4   POST POST OP 5 6 - *
2   POST 5 6 - *
3   POST POST 6 - *
3   POST POST POST - *
4   POST POST POST OP *
2   POST POST *
4   POST POST OP
2   POST
1   EXPR
```

Derivation 6.9: Bottom-up right-most Derivation of the expression 3 4 + 5 6 - *

We now have four derivation forms: top-down left-most, top-down right-most, bottom-up left-most, and bottom-up right-most. The first and last of these are the most common, and the ones most often used by compilers.

One last thing to note: If you were asked to give a derivation of an expression using our postfix grammar, there would be many correct answers. You could chose which order to invoke productions and which non-terminal to replace in each step. But if any of the four flavors mentioned in the previous paragraph were requested, there would be exactly one correct answer. There would be no choice as to which non-terminal to replace or which production to invoke.

The only-one-right-derivation property is true of this grammar because it is non-ambiguous. Ambiguous grammars will be described in Section 6.3.

6.2.2 Syntax Trees

Derivations can be represented graphically as syntax trees. In a syntax tree, the children of each non-terminal are the items from the right hand side of the production that was used to replace the non-terminal. Figure 6.1 shows the syntax tree for the derivation in Derivation 6.6. Note that if the derivation is given in tree form, there is no sense of left-most or right-most because the tree gives no indication of what order the productions were invoked. For the derivation represented in Figure 6.6, it would not matter what order the productions were invoked. All derivations would produce an identical tree. The only-one-tree property is true because the grammar used for this derivation is non-ambiguous. Ambiguous grammars are discussed in the next section.

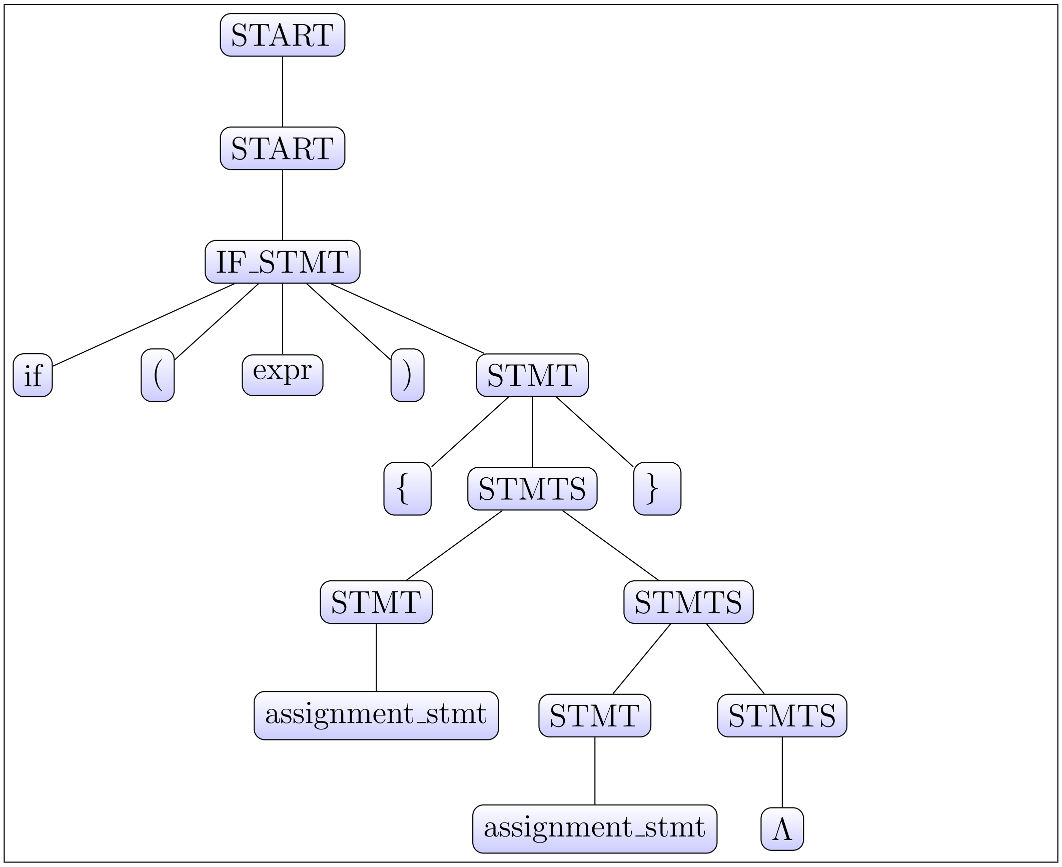


Figure 6.1: A syntax tree representing the derivation in Figure 6.6.

6.3 Ambiguous Grammars

English, and I presume all spoken languages, is ambiguous. A single statement can be interpreted different ways. Consider the two statements

> Fruit flies like an apple. Time flies like an arrow.

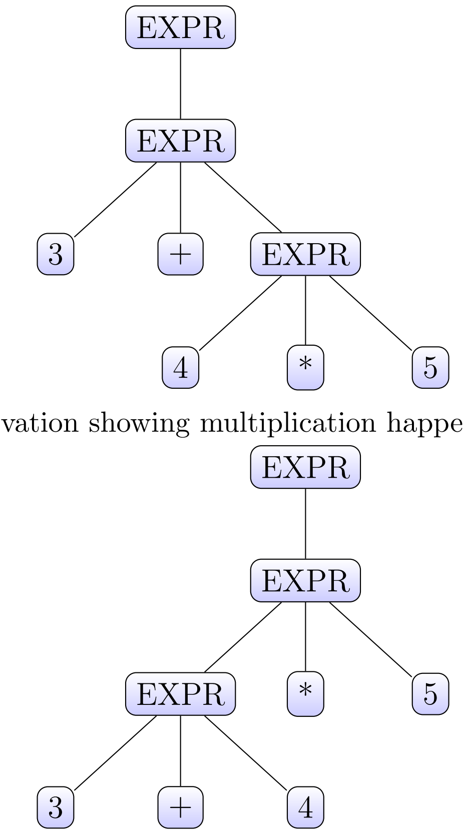
Both statements appear structurally the same; only a couple of words were changed. However, with the normal interpretations, "flies" is the subject in the first sentence, but the verb in the second (unless the second sentence is referring to flies native to the Tardis). If we are sufficiently familiar with the English language, we come up with the correct interpretation of each sentence without even consciously thinking about it. But what if you tried to write a program to understand English? How would it distinguish betweeen the two.

Consider the statement, "I shot the man with the gun." When the police show up and you utter that sentence, you hope they interpret it as, "See that man over there with a gun? I shot him in self defence." You hope they don't interpret it, "See this gun in my hand? I shot that [possibly unarmed] man over there with this gun." Although I hope you never find youself in a situation where you have to utter such a sentence, if you do, you might want to pick a less ambiguous statement.

A grammar is ambiguous when a single input can produce multiple different meanings. For CFGs, the "meanings" are defined by the syntax tree produced by parsing a statement. The grammar in CFG 6.10 illustrates this. The expression $3 + 4 * 5$ can be derived two different ways. One has the multiplication happening first and the other with the addition happening first. These two derivations are illustrated by the trees in Figure 6.2.

```
EXPR → EXPR + num
EXPR → EXPR * num
EXPR → num
```

CFG 6.10: An ambiguous expression grammar.



(a) Derivation showing multiplication happening first.

(b) Derivation showing addition happening first.

Figure 6.2: Two syntax trees showing derivations of $3 + 4 * 5$ using the ambiguous grammar in CFG 6.10

Note that the ambiguity of a grammar is defined by the existence of multiple trees for the same expression. It is not defined based on different derivation orders for the same expression. Recall that when a derivation is represented as a tree, the top-down vs. bottom-up and left-most vs. right-most nature of the derivation is obscured. Any grammar can be used in a left-most or right-most (or middle-most) fashion. That does not mean the grammar is ambiguous (otherwise, all grammars would be). If each of these derivations results in the same tree, the grammar is considered non-ambiguous.

For programming languages, ambiguous grammars are clearly bad. The expression $3 + 4 * 5$ should not evaluate to a different value simply because the compiler did a left-most derivation versus a right-most derivation. There are expression grammars that resolve the ambiguity of the grammar in Figure 6.10, but since this is a theory book, we will leave that topic for a compiler book.

6.4 Grammar Forms

One of the tasks of a compiler is to find a derivation of the input (the source code being compiled) given the definition of the language (typically expressed as a CFG). This part of a compiler is known as a parser. In order for a parser to be efficient, there needs to be turn-the-crank algorithms for finding a derivation. The algorithms should not require a guess-and-check process. In particular, given the next N input symbols, the parser should always know what production to use next in the derivation.

There are two primary algorithms used by parsers. One is a top-down algorithm, the other is a bottom-up algorithm. The top-down algorithm starts with the start symbol and based on the next input token invokes a production. This continues until all input is exhausted and only terminals remain. The bottom-up algorithm reads tokens and periodically reduces some collection of symbols that match the right hand side of a production to the symbol on the left hand side of that production. This continues until all input is exhausted and only the start symbol remains.

There are two grammar forms that are used in support of these compilers. An LL(1) grammar reads input Left to right (the first L), does a Left-most derivation (the second L), and can always make a decision with 1 token of look-ahead. An LR(1) grammar reads input Left to right, does a Right-most derivation and can always make a decision with 1 token of look-ahead. LL(1) grammars are used by top-down parsers, and LR(1) grammars are used by bottom-up parsers.

These grammars are related as follows.

$$LL(1) \subset LR(1) \subset CFG$$

Note that these are strict subsets. There are things that can be done with an LR(1) grammar that cannot be done with an LL(1) grammar. An there are things that can be done with CFGs that cannot be done with an LR(1) grammar.

To illustrate what causes difficulties with the top-down algorithm (and therefore the restrictions placed on LL(1) grammars), consider the grammar given in CFG 6.11. This grammar defines fully parenthesized expressions.

```
1 EXPR → number
2 EXPR → (EXPR + EXPR)
3 EXPR → (EXPR - EXPR)
4 EXPR → (EXPR / EXPR)
5 EXPR → (EXPR * EXPR)
```

CFG 6.11: This CFG defines a fully parenthesized expression.

Since they are fully parenthesized, the grammar is unambiguous. However, with this grammar, how many tokens of look-ahead are required to determine which production to invoke first? If the first token is a number, then clearly Production 1 must be invoked. But if the first token is an open parenthesis, then how many additional tokens must be read to determine which of the productions 2-5 should be invoked? Three is not enough because expressions can be nested as in

(((5-3)-3)*2)

As a result, this grammar is not an LL(1) grammar.

The specifics of the top-down and bottom-up algorithms are left to a course on compilers. Also, the algorithms for transforming a grammar into LL(1) or LR(1) form are left to a course on compilers. Later, we will look at other grammar forms and transformations to convert a grammar into a particular form, but we will be interested in a form that is useful for proving properties of grammars, not a form that is useful to a compiler.

6.5 Exercises

For the following problems, assume $\Sigma = \{a \ b\}$ unless specified otherwise.

1. Write a CFG for PALINDROME.

2. Write a CFG for the language defined by the regular expression a^*b.

3. Write a CFG for arithmetic expressions in postfix notation (where the operators come after the operands).

4. Give a derivation for the program in Listing 6.4. Be sure to list the production number used in each step.

5. Provide derivations for the following two statements given the following CFG. Be sure to list the production numbers for each step.

   $3 + 4 * 5 + 7$
   $(3 + 4) * (5 + 7) * 8$

```
1 START → AE
2 AE → AE+AE
3 AE → AE*AE
4 AE → (AE)
5 AE → number
```

6. Use the following CFG and show derivations for the two expressions from the previous problem. Be sure to list the production numbers for each step.

```
1 START → EXPR
2 EXPR → EXPR + TERM
3 EXPR → TERM
4 TERM → TERM * FACT
5 TERM → FACT
6 FACT → (EXPR)
7 FACT → number
```

7. Is the language defined by CFG 6.3 ambiguous? If so, show two syntax trees for the same input. If not, you must argue that it is not possible to create two different syntax trees for the same input.

8. Is the grammar in Question 5 ambiguous? Explain.

9. Is the grammar in Question 6 ambiguous? Explain.

10. Do the grammars in Questions 5 and 6 define the same language? Explain.

Chapter 7

Pushdown Automata

When talking about regular languages, we found that an existing concept, state machines (aka finite automata), proved very useful for processing regular languages. In this chapter, rather than showing that an existing machine is useful for processing context free languages (those that can be defined by a CFG), we will instead invent such a machine.

Finite automata have no way of "remembering" what path they took. This is why they were unable to solve the balanced parenthesis problem. They had no way to remember how many opening parenthesis they encountered, so they couldn't enforce that there be the same number of closing parenthesis. To solve this problem, we propose that a push-down stack be added to finite automata thus making a new machine called a pushdown automata.

To illustrate how this could allow us to solve the balanced parenthesis problem, suppose each time we encountered an opening parenthesis in the input, we pushed it onto the stack. Each time we encountered a closing parenthesis, we popped an open from the stack. If we never attempted to pop an empty stack, and if at the end of the input, the stack was empty, then we must have only encountered balanced parenthesis.

For pushdown automata (PDA), the input is viewed as residing on a tape. Each "cell" on the tape contains a single character (or token). The tape is infinite in length, and every call after the end of the input contains a Λ. Each time the tape is read, it advances to the next cell. Note that with this configuration, reading past the end of the input always returns a Λ.

The stack for a PDA is also viewed as a tape, infinite in both directions, with all cells initially holding a Λ. When a token is pushed onto the stack, it is written to the tape and the tape advances to the next cell. When a token is popped off the stack, the tape is advanced to the previous cell and the token at that location is read. Note that with this configuration, popping an empty stack always returns a Λ.

It should be pointed out, that a Λ in a PDA is very different from a Λ in an FA. For an FA, a Λ transition means you go to the next state without reading any input. For a PDA, it means you're read past the input of popped below the bottom of the stack.

The other difference between an FA and a PDA, is that with an FA, transitions imply reading input. For a PDA, each read is made explicit.

Because of the differences between FA's and PDA's, we need a new set of symbols. Figure 7.1 shows several of these. The start and end states are rounded rectangles. The end states make it clear whether the input is accepted or rejected. The explicit reads are diamonds, and the outbound edges are labeled with the token that was read from the input tape.

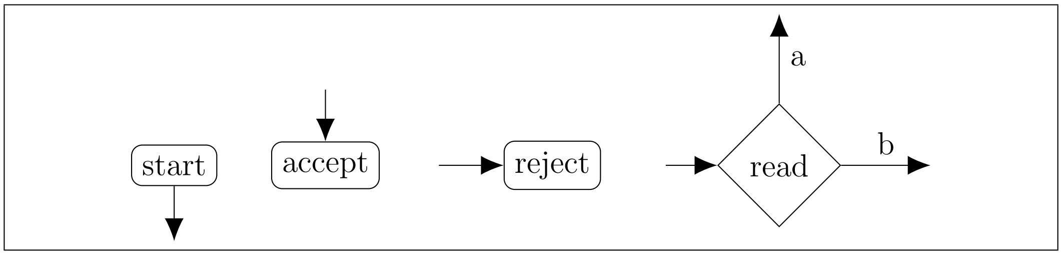
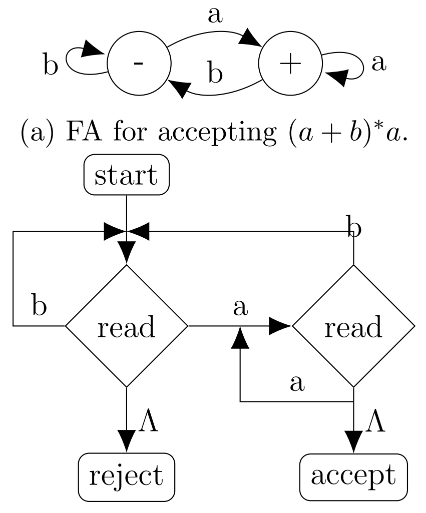


Figure 7.1: Symbols for PDAs. The start symbol is only allowed to have outbound arrows. The accept and reject symbols are only allowed to have inbound arrows. The is no requirement on where the arrows enter or leave the symbols, only that the diagram be neat. When the PDA enters a read symbol, the input tape is read and the outbound arrow corresponding to the read symbol is followed.

It is possible to draw an FA using the PDA symbols. An example is given in Figure 7.2. It should be clear from this example that any DFA can be drawn using the new symbols. However, the purpose of the new symbols isn't simply to give us another wan to draw DFAs. Instead, the new format allows us to add memory in the form of a pushdown stack. Push operations are encoded in a rectangle. Pop operations use the same symbol as read, but with the word “pop” instead of “read”.



(a) FA for accepting $(a+b)^*a$.

(b) PDA for accepting $(a+b)^*a$.

Figure 7.2: Both an FA and a PDA for accepting all strings that contain a double letter.

Figure 7.3 shows a PDA for accepting a^nb^n. Each time an a is read, it is pushed on the stack. Each time a b is read, the stack is popped. If the stack is empty, the string is rejected because we had too many b's. If we read an a following a b, we also reject the string.

Since Figure 7.2 shows a solution to to a^nb^n, it is clear that PDAs are more powerful than FAs.

This section needs work.

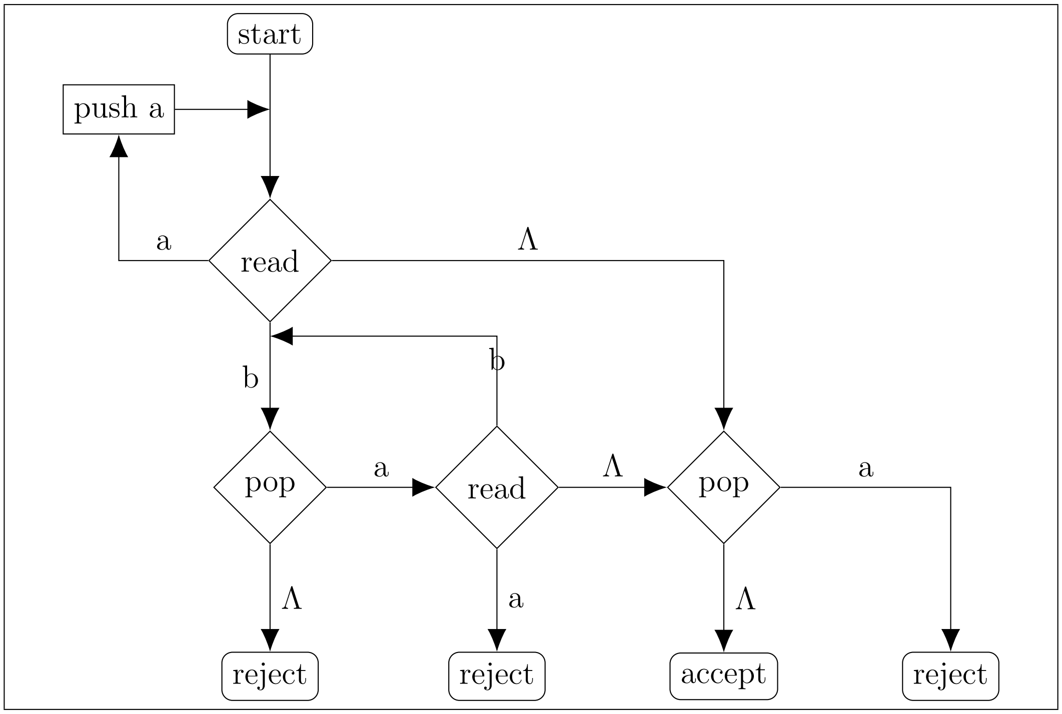


Figure 7.3: PDA for accepting $a^n b^n$.

Chapter 8

Theory of Context Free Languages

In this chapter we will examine various properties of context free languages.

8.1 All regular languages are context free

When proving Kleene's Theorem, we used various constructions to show the equivalence of various forms of expressing a language. We will do the same here.

Take any FA, and give each state a unique name. To build an equivalent CFG, create a non-terminal for each state in the FA. For each edge in the FA, create a production as illustrated in Figure 8.1. For every final state in the FA, create a production with that state on the left-hand side and Λ on the right. For a complete example of using this construction, consider the FA in Figure 8.2. This FA is represented by the following CFG.

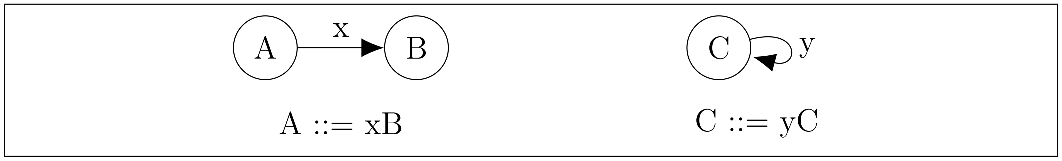


Figure 8.1: Creating CFG productions from FA transitions.

```
S → aB
B → bD
B → aC
C → aC
C → Λ
D → aB
D → bE
E → Λ
```

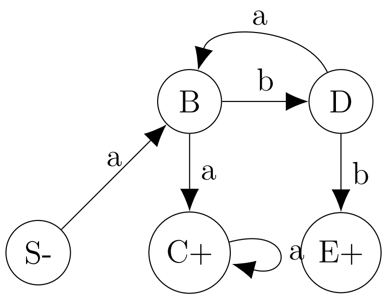


Figure 8.2: Illustration of FA that is turned into a CFG.

This section needs work. Need to do a proof that this works.

8.2 Chompsky's Normal Form

In this section we show that all grammars (for languages that don't include the empty string) can be transformed into Chompsky's Normal Form (CNF). In this form, all productions either have exactly two non-terminals on the right hand side (and no terminals) or exactly one terminal.

8.2.1 Eliminating Lambda Productions

CFG 8.1 shows a CFG that includes a Λ production. CFG 8.2 shows an equivalent CFG without the Λ. There are two differences in the second CFG: the production for L does not include the Λ option, and there is a second production for S, which does not include the L. This additional production is necessary because in the original CFG, L can be replaced with Λ (nothing).

Since that option was removed in the second CFG, we need an additional production that doesn't include L.

```
S → STUFF L OTHER
...
L → Λ | a | b
```

CFG 8.1: CFG that includes a Λ production.

```
S → STUFF L OTHER
S → STUFF OTHER
...
L → a | b
```

CFG 8.2: Equivalent CFG without a Λ production.

This transformation seems easy enough, but there is a potential problem. What if removing the Λ from one production adds a Λ to another. Consider the grammar in CFG 8.3. If we follow the pattern from the previous example using this grammar, when we eliminate the Λ from the A production, we will introduce a Λ to the B production. When we eliminate that Λ, we will re-introduce the Λ to the A production.

```
S → STUFF
...
A → Λ | B
B → STUFF | A
```

CFG 8.3: CFG that includes recursive Λs.

The solution to recursive Λs is to eliminate them all at once. The first step is to find all non-terminals that are nullable. That is, all non-terminals that can be replaced by a Λ either directly or indirectly. A non-terminal is nullible if either:

$\texttt{X} \rightarrow \Lambda$
or

X → ... → Λ

We can find all the nullables using the following procedure:

1. Mark all non-terminals with Λ productions.
2. If a production has only marked non-terminals on the right hand side, mark the non-terminal on the left hand side.
3. If Step 2 resulted in marking a new non-terminal, mark that non-terminal throughout the grammar.
4. Repeat steps 2 and 3 until no new non-terminals are marked.

All terminals marked as a result of this procedure are nullable. Consider the following grammar (which happens to be for the language $(A+b)^*bb(a+b)^*$):

S → X Y
X → Z b
Y → b W
Z → A B
W → Z
A → a A | b A | Λ
B → B a | B b | Λ

The productions for both A and B include Λ, so cross out every occurrence of A and B. This results in the following grammar:

S → X Y
X → Z b
Y → b W
Z → ~~A~~ ~~B~~
W → Z
~~A~~ → a ~~A~~ | b ~~A~~ | Λ
~~B~~ → ~~B~~ a | ~~B~~ b | Λ

Since Z has a production where everything on the right hand side is crossed out, cross out all occurrences of Z. This results in the following grammar:

S → X Y
X → ~~Z~~ b
Y → b W
~~Z~~ → ~~A~~ ~~B~~
W → ~~Z~~
~~A~~ → a ~~A~~ | b ~~A~~ | Λ
~~B~~ → ~~B~~ a | ~~B~~ b | Λ

Since W has a production where everything on the right hand side is crossed out, cross out all occurrences of W. This results in the following grammar:

S → X Y
X → ~~Z~~ b
Y → b ~~W~~
~~Z~~ → ~~A~~ ~~B~~
~~W~~ → ~~Z~~
~~A~~ → a ~~A~~ | b ~~A~~ | Λ
~~B~~ → ~~B~~ a | ~~B~~ b | Λ

This didn't result in any additional non-terminals being crossed out, so we are done. The nullables are A, B, W, Z.

Now that we have a procedure for identifying all the nullables, we can give a procedure for eliminating Λ productions. The procedure is as follows:

1. Identify all nullables.

2. Delete all Λ productions.

3. For all productions with a nullable on the right hand side, create new productions for all subsets of the nullables, but don't introduce any new Λ productions nor duplicates.

To illustrate this procedure, consider the following CFG:

S → a | X b Y | a Y a
X → Y | Λ
Y → b | X

The nullables are X and Y. The second and third S productions include nullables, so we need to create subsets. Take the production that has $X\ b\ Y$ on the right hand side. We will need new productions without the X, without the Y, and with neither. The production with $a\ Y\ a$ on the right hand side will need a new production without the Y. This yields the following CFG:

1. S → a
2. S → X b Y | b Y | X b | b
3. S → a Y a | a a
4. X → Y
5. Y → b | X

Production 2 shows the second S production and the new ones created from the subsets. Production 3 shows the third S production and the new one created from the subsets. Productions 4 and 5 contain nullables, but a subset from those productions would create new Λ productions, so no new productions are created from these.

Note that for Production 2, b is not eliminated in the subset process because it is not nullable (as terminals never are).

8.2.2 Eliminating Unit Productions

A unit production is a production with a single non-terminal on the right hand side. Note: It must be a single non-terminal. A production with a single terminal is not considered a unit production.

This procedure is similar the the procedure for eliminating Λ productions, and like the Λ procedure, we need to be aware of recursive unit productions. The procedure is as follows:

For every pair of terminals A, B for which either:

> A → B
> A → $\mathtt{X}_1$ → $\mathtt{X}_2$ ... → B

and the non-unit productions of B are

> B → $S_1|S_2|S_3$

create new productions for A:

> A → $S_1|S_2|S_3$

Do all pairs simultaneously, and don't create any new unit productions.

To identify all the unit pairs, start by making a list of all unit productions. Then invoke unit productions on the right hand side. Eliminate duplicates and any productions of the form A → A. Repeat this until no new unit pairs are identified.

This process is illustrated starting with the following CFG:

1. S → EXPR
2. EXPR → EXPR + TERM
3. EXPR → TERM
4. TERM → TERM * num
5. TERM → num

Productions 1 and 3 are unit productions. Production 5 is not a unit production because even though the right hand side has only one symbol, it is a terminal, not a non-terminal. We can invoke production 3 on the right hand side of Production 1 yielding the following unit pairs:

S → EXPR
S → TERM
EXPR → TERM

The non-unit productions are:

```
EXPR → EXPR + TERM
TERM → TERM * num
TERM → num
```

Creating the new productions yields the following CFG:

1. `S → EXPR + TERM`
2. `S → TERM * num`
3. `S → num`
4. `EXPR → EXPR + TERM`
5. `EXPR → TERM * num`
6. `EXPR → num`
7. `TERM → TERM * num`
8. `TERM → num`

8.2.3 Eliminating mixed productions

A mixed production is one that includes both terminals and non-terminals on the right hand side. To eliminate all mixed productions, create new non-terminals for each terminal then in each mixed production, replace the terminal with the corresponding non-terminal.

Starting with the CFG we had at the end of the previous section, we need new terminals for $+ \ * \ num$. These are as follows:

```
PLUS → +
TIMES → *
NUM → num
```

Substituting these into the previous grammar we get:

1. S → EXPR PLUS TERM
2. S → TERM TIMES NUM
3. S → num
4. EXPR → EXPR PLUS TERM
5. EXPR → TERM TIMES NUM
6. EXPR → num
7. TERM → TERM TIMES NUM
8. TERM → num
9. PLUS → +
10. TIMES → *
11. NUM → num

Note that we did not perform the substitution in productions 3, 6, or 8 because they are not mixed. They contain only a terminal. If we had performed the substitution on these productions, we would have created new unit productions. We went through a bunch of trouble to eliminate them, so we don't want to create new ones.

8.2.4 Conversion to Chompsky's Normal Form

The final step in converting to Chompsky's Normal Form is to transform all productions with more than two non-terminals on the right hand side into productions with exactly two. Consider the following production:

```
A → B C D E
```

We will take the first two non-terminals on the right hand side and create a new non-terminal to replace them. We'll then repeat this until we are left with just two non-terminals. The first two are B C, so we will create a new non-terminal Z to replace them.

```
A → Z D E
Z → B C
```

We will then replace $Z\ D$ as follows:

```
A → Y E
Z → B C
Y → Z D
```

It might be tempting to try to optimize the process by finding the minimum number of new non-terminals, but there is no need for this optimization. Simply start with the first production with three or more non-terminals and create a new non-terminal for the first two on the right hand side. Continue this process until all productions have exactly two non-terminals or a single terminal.

8.2.5 Why Chompsky's Normal Form?

The obvious question is, "Why would we convert a grammar to Chompsky's Normal Form?" The answer is that in practice we don't. We simply needed to prove that it could be done. The reason why this is interesting is that a grammar in Chompsky's Normal Form (CNF) will always produce binary parse trees. Since they are binary trees, we can define the relationship between the length of an input and the minimum tree depth needed to generate a string of that length.

In the previous sections, we showed that any CFG (that didn't allow empty strings) could be converted to CNF. To eliminate the exception, we could simply add a production S → Λ to our CNF grammar. Adding this production does not violate the binary tree nature of derivations.

8.3 The pumping lemma for CFGs

This section needs work. zzz

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.