A Very Simple Compiler Book

Philip W. Howard

September 16, 2019

Contents

List of Figures

Preface

I've been teaching Introduction to Grammars (aka Computer Theory) and Compiler Methods at Oregon Institute of Technology for several years. There is a Computer Theory text that I really like, but it has become criminally expensive (as I'm writing this, Amazon would be glad to sell you a new paperback edition for over $300). Although there are good compiler books available, none of them seem to approach the subject they way I do. I have no visions of preparing the next generation of compiler masters. There are plenty of reasons to study compilers even if you're never going to write one. My course focuses on those reasons.

So, why not write my own? Lot's of reasons that I won't enumerate, but obviously I've attempted to get past those and actually write the book. As I've written this, I've wandered between "Should it be a grammars book?" and "Should it be a compilers book?" and "Should it be both?".

It is possible to write a good grammars book without dealing with compiler issues. The best books I've seen on this subject do this. But the program at OIT is a very hands-on, so my students always want an answer to "Why do we study this?". Compilers is certainly part of the answer (though I do my best to convince my students that there are other reasons as well). So my grammars book should mention some of the compiler issues.

I can't imagine a good compiler book that doesn't touch on some of the concepts normally covered in a grammars book. Where would a modern compiler be without regular expressions and context free grammars?

So I'm attempting to write a book that will be useful in both classes. The first part of the book deals with the theoretic issues: The Theory of Computation (aka Formal Languages). This section is structured in such a way that the parts that are necessary for a compiler course can be easily segmented out. The later parts, the compiler parts, contain references back to the relevant sections of the theory part.

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

Computer Theory aka Formal Languages

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

1.2 Formal Languages

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

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

If you attempted to create a Language Processor for English, it should probably reject statements like, "Cup sky red Perl". But could you create a

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.

processor that accepts all valid English (including English poetry), and rejects all non-English. To get a sense of the difficulty of this challenge, I refer you to poetry (considered valid English poetry by those who get to decide such things) by ee cummings (capitalization is correct).

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

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

Formal languages consist of the following:

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

Let's illustrate with an example.

$\Sigma = \{ab\}$

All strings of five letters.

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

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

1. Recursive definitions (not to be confused with Recursive Languages discussed in Chapter 6)
2. Regular Expressions
3. Context Free Grammars

Chapter 2

Recursive Definitions

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

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

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

Consider the following non-recursive definition of arithmetic expressions:

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

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

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

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

How about:

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

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

Let's instead give a recursive definition:

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

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

$(2 + 7)/5$

The construction is as follows:

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

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

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

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

Let's generalize it to the statement:

> Valid statements in AE never have empty parenthesis.

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

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

The proof is as follows:

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

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

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

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

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

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

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

2.1 Constructing Recursive Definitions

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

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

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

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

2.2 Summary

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

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

Add practice problems here.

Chapter 3

Regular Expressions

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

Regular expressions include the following features:

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

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

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

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

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

The most common enhancements to this syntax are as follows:

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

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

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

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

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

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

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

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

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

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

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

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

Exercises

1. Write a regular expression for a string containing any odd number of the letter `a`.
2. Write a regular expression for C (or Java) variable names. Valid characters include upper and lower case letters, digits, and the underscore (_).
3. Write a regular expression for a string containing any number (including zero) of a positive even number of `a`'s followed by an add number of `b`'s. The following are valid strings: `aaaabbb`, `aabaabaabbb`, `aaaaaabbbbbaabaabaab`. The following are not valid strings: `aaab`, `aaaabbbaa`, `bbbaab`.
4. For the previous question, state why each of the non-valid strings are non-valid.
5. Write a regular expression for a floating-point constant. The following rules apply:
   (a) The integer part cannot have leading zeros unless the integer part is zero.
   (b) If there is a decimal point, it must be followed by at least one digit.
   (c) The decimal part must not have trailing zeros unless the decimal part is zero.

3.1 Language Families

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

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

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

So we now have three separate but related languages:

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

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

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

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

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

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

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

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

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

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

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

Theorem 1 *All finite languages are regular.*

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

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

Chapter 4

Finite Automata

Chapter 5

Context Free Grammars

This section needs work.

Chapter 6

Beyond Context Free Languages

This section needs work.

Part II

Overview

Chapter 7

Introduction

7.1 Why?

The first question that many prospective compiler students have is, "Why should I study compilers?" The following answers come to mind:

1. They're cool. Why wouldn't you want to study compilers?

2. It gives you a chance to understand what your primary tool does. It would seem strange for a carpenter to not know what a hammer does, or to not know the trade-offs between a hammer and a nail gun. Granted, a carpenter doesn't need to know the physics of how these tools work, but they should have a general idea.

   Many software developers never consider their compilers. They have this large blob of software known as an "IDE". They type specialized text into their IDE and click various buttons, and if they are luck, they get a working program. My claim is that better programmers have a better understanding of their tools. This allows them to make better use of these tools, and thus better use of their time, and thus (sometimes) more of their employer's money.

3. It gives you a chance to manage a large code base. I've heard of undergraduate computer science students who graduate having never managed anything bigger than a two week lab. In my ten week compiler course, students write an entire compiler composed of tens of source files and thousands of lines of code. Design and implementation decisions made early have strong consequences later in the term. This is a valuable experience that every developer should have before they step into a job claiming they know what they're doing.

   Another advantage of the large project is that it allows the introduction of testing techniques that are harder to make "real" in smaller environments.

4. Applied data structures. A compiler has a lot of data that needs to be managed. Many data structures present "clean" data structures, but real-life data-problems often aren't so clean. How about a cyclic tree that contains links to a stack of hash tables that in turn has links back to nodes of the tree? OK, if your tree is cyclic, you probably have a bug, but most data structures classes don't give you the opportunity to make (and then fix) such bugs. Having the opportunity to get buried in data is good for you.

5. Applied algorithms. I started to learn calculus in high school. I continued my learning of calculus in college. But I didn't really understand what a dx was until I encountered them in my calculus based physics class. They were no longer merely symbols on a paper, they were now applied, and that's how I finally grasped what they really were. In the same sense, a compiler allows you to apply quite a number of algorithms (and data structures). In doing so, they should be more "real".

6. Performance considerations. Computers have gotten so fast, we rarely have to worry about performance (time or memory) considerations. Most compiler runs are also fast enough that the compiler's performance doesn't much matter. But given a compiler, you can compile something very big like the Linux Kernel. Performance now matters. Because of this, a compiler class gives the right to talk about performance considerations in a practical sense.

7.2 What is a compiler?

Perhaps before asking, "Why study compilers?" it would have been best to establish what a compiler is. But most students, by the time they get to a compiler course, have already used compilers so they have some sense of what they are: The software that translates my source code into executable code. We need to be a bit more formal as well as a bit broader in our definition.

In the broadest sense, a compiler takes code as input and outputs an improved version of that code. For the compiler to be valid, it must follow two guiding principles:

1. The compiler must preserve the meaning of the original program.
2. The compiler must improve the code in some discernible way.

Figure 7.1: A compiler is a program that takes code as input and outputs an improved version of that code.

To illustrate the first principle, if you had a program that output the numbers from 1..10 and the compiler transformed it so that it output one of e.e. cummings's poems, then clearly the compiler did not preserve the meaning of the original program.

The second principle is a bit less clear. If the code is a program, and you want to execute your program, then transforming your source code into an executable program would be an improvement. But suppose your source was a C program and the output was an equivalent Java program. Would that be an improvement?

This definition of a compiler is deliberately broad. The preprocessor for a C compiler (often a separate program from the C compiler) is considered a compiler. LaTeX, which was used to typeset this book, is a compiler. Indeed, any converter program can be considered a compiler given this definition. For the sake of this book (and most compiler courses), we will primarily

restrict ourselves to compilers where the input is a programming language. The output, however, does not need to be executable code. It can be any form of "improved" code.

Chapter 8

Compiler Structure

Compilers are typically broken into two broad pieces: the front-end and the back-end. The frond-end deals with language specific issues. The back-end deals with target machine specifics.

A C compiler would need a very different front-end compared with a Haskell compiler because the syntax of the languages is very different. One way to view the front-end of the compiler is that it answers the question, "Is this a valid program?" If you feed a Scheme program into an R compiler, the answer would certainly by "No". If a Java program that was missing a semicolon was fed into a Java compiler, the answer would also be, "No", but for very different reasons.

The back-end of a compiler deals with target specific issues. The back-end for a compiler that targets the Java Virtual Machine (JVM) would be different than the back-end for a compiler that targets and ARM processor. Both of those would be different from a C preprocessor that simply outputs text.

Separating the front-end from the back-end makes sense because they are logically quite different, but it also makes sense because if they are properly designed, you can mix and match different front-ends and back-ends. Suppose you had a really good C compiler, but you wanted to port it to target a different machine. There would be no reason to rewrite the front-end. The C language is the same no matter what machine you are targeting. So if you

wrote a new back-end and attached it to your existing front-end, you would now have a new compiler. Mixing can go the other way as well. Starting with the same really good C compiler, suppose you wanted a compiler for a different language (such as Go). There is no reason to rewrite the back-end, simply create a front-end for the new language and attach it to your existing back-end, and you have a new compiler.

For this mixing and matching to work seamlessly, there needs to be a common interface between the front-ends and the back-ends. This interface is typically called an Intermediate Representation (IR) — see Chapter 9. The front-end translates the language into the IR, and the back-end translates the IR to the target representation. This is illustrated in Figure 8.1

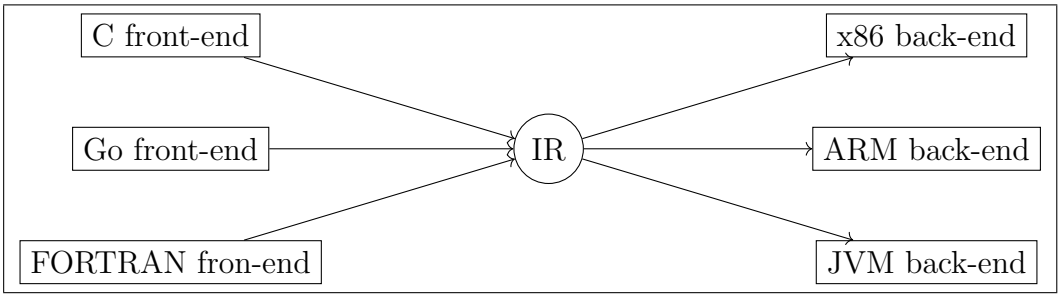


Figure 8.1: A properly structured compiler allows you to have multiple language-specific front-ends joined to multiple target-specific back-ends. The two a joined using an intermediate representation.

Compiler front-ends typically contain two components: A scanner (aka tokenizer or lexer), and a parser. Scanners are discussed in detail in Part III and parsers are discussed in Part IV.

Most computer program source code comes in the form of a stream of characters (a text file). The scanner turns the stream of characters into a stream of tokens. For example, it the scanner read the characters "`some_variable_name`", it would output the single token `IDENTIFIER`. For a longer example, the string "`int main() { printf("hello world"\n); }`" would be converted to the tokens "`TYPE IDENTIFIER ( ) { IDENTIFIER ( STRING_LIT ) ; }`".

Note: Programmers normally format their code to make it "pretty". The code in the above examples was not formatted to emphasize that to a compiler, code is just a stream of characters.

The parser takes a stream of tokens and decides whether that stream of tokens makes up a valid program (more specifically a "compilation unit") in the language. The parser only looks at the syntax of the language. As a result, "valid program" only means "a program that is syntactically correct". It does not mean "a bug free program".

Chapter 9

Intermediate Representations

Part III

Scanning

Chapter 10

Creating a Scanner from Regular Expressions

To specify a scanner for a compiler, a developer could specify a regular expression for each token type. But how do you go from that specification to code that can actually process the input? This chapter discusses the algorithms used to convert regular expressions into a scanner. The general process is to convert the regular expression into a finite automaton. The finite automaton can then be evaluated using a simple table look-up on each character.

10.1 Finite Automata

Finite Automata (aka State Machines) consist of a finite number of states and transitions between states. One state is defined as the start state, and any number of states can be defined as final states. They are often drawn as diagrams as illustrated in Figure 10.1.

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.

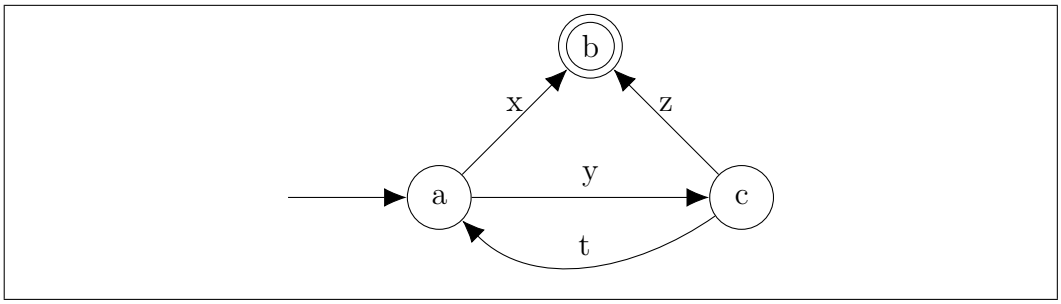


Figure 10.1: This is a sample FA. Node a is the start state (it has an incoming arrow from nowhere). State b is a final state (signified by the double-circle). The transitions are labeled by what letter is used to move from one state to another.

Using the FA in Figure 10.1, and the input ytx, the processing starts in State a (the start state). The y is used to transition to State c. The t is used to transition back to State a. The x is used to transition to State b. Since the input is exhausted while in a final state, the string is accepted by the FA.

Two conditions can cause a string to be rejected:

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

Given these rules for processing FA's, it should be clear that the FA in Figure 10.1 is equivalent to the regular expression `(yt)* (x | yz)`

10.1.1 Non-deterministic Finite Automata

The FA illustrated in Figure 10.1 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 10.2.
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.

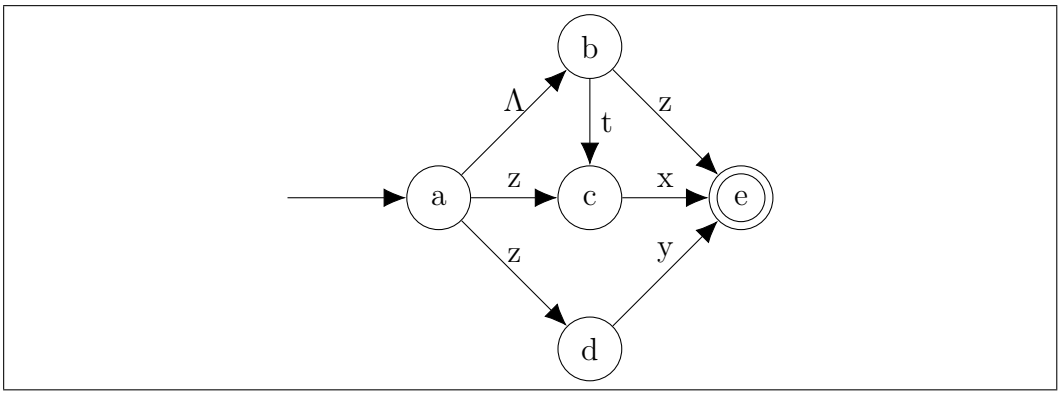


Figure 10.2: This is a sample NFA. Node a has multiple outbound transitions on z. It also has an outbound transition on Λ meaning you could leave Node a without consuming any characters.

The NFA in Figure 10.2 accepts the following stings:

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

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

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

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

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

10.2 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[2]. The beauty of Thompson's construction is that it is a mechanical process – one that doesn't require any creative thought. In other words, it can be automated. A computer program can be written to perform this conversion.

If two FA's each have a single start state and a single final state, and if the start state doesn't have any inbound edges and the final state doesn't have any outbound edges, then the two FA's can be composed by connecting the end state of one FA to the start state of the other using a Λ transition. Thompson's Construction makes use of this fact by creating composed FA's for each operation supported by regular expressions (concatenation, alternation, and Kleene Closure). An NFA can be built for any regular expression simply by composing it one small piece at a time using Thompson's three diagrams. Figure 10.3 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 10.3.

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

[1]Section Figure 10.3 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.

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

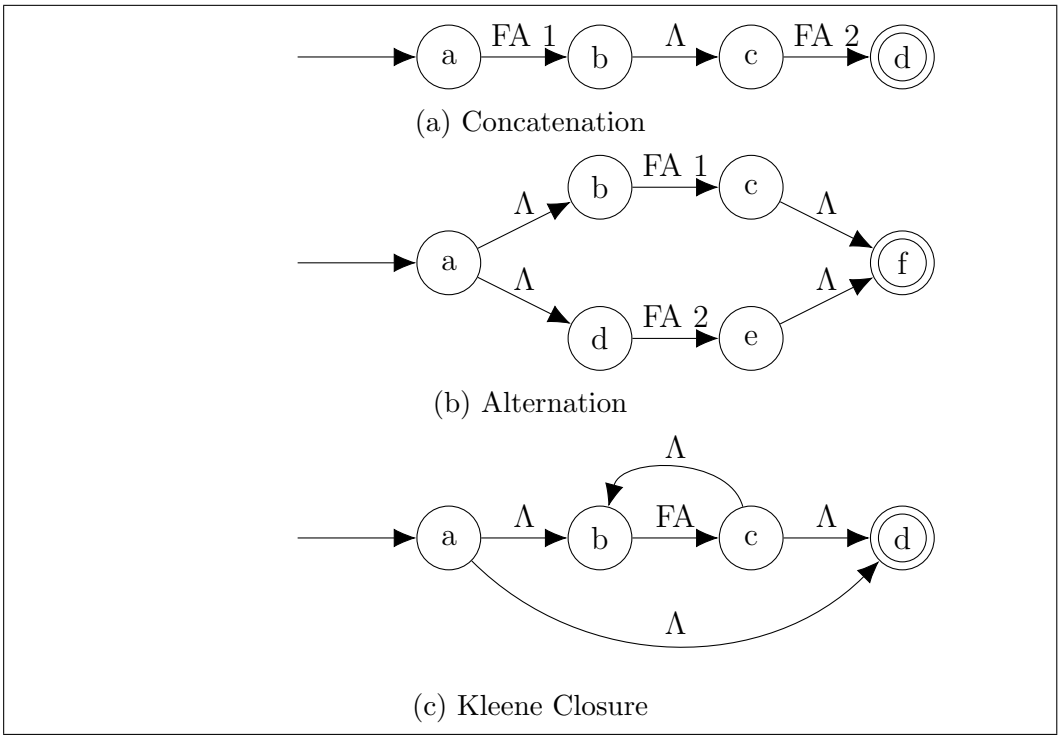
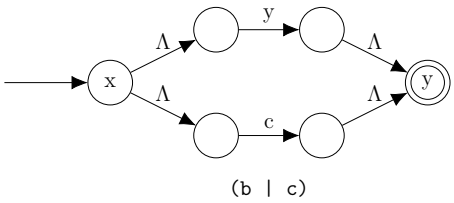


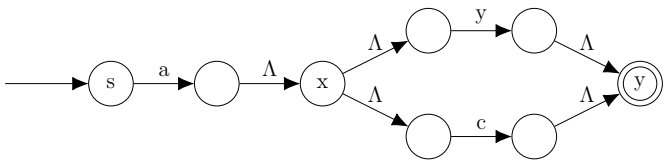
Figure 10.3: Thompson's Construction method makes use of these three diagrams. For each diagram, the FA(s) being composed are represented as two nodes (the start and end nodes) labeled FA, $FA1$, or $FA2$.

`(b | c)`) and work out. So first draw the alternation diagram as shown below:



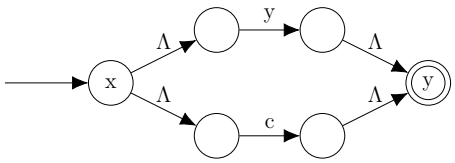
The resulting diagram gets dropped into the `FA 2` position of the concatenation diagram as illustrated below. Note how the node labeled x in the

above diagram is in the location of the node labeled c in the concatenation diagram, and the y node is in the d position.



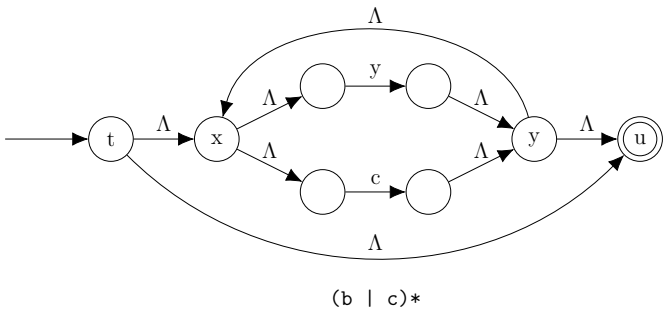
`a (b | c)`

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



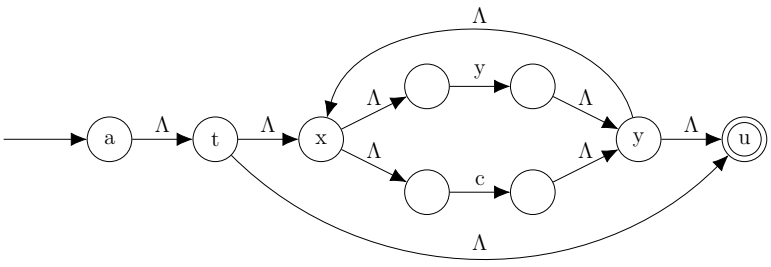
`(b | c)`

This is now dropped into the Kleene Closure construction:



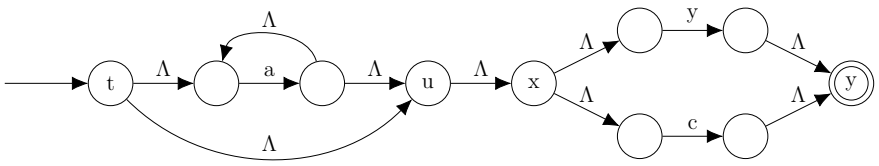
`(b | c)*`

Finally, we drop this into the concatenation construction:



a (b | c)*

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



a* (b | c)

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

10.3 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 10.4.

In this figure, each node is labeled so they can be explicitly referred to. For each state, we can ask the question, "What states could I wind up in if I encounter a particular letter in the input?". For example, suppose we haven't consumed any input yet, what states could we be in? Clearly we could be in State 1, the start state, but because of the Λ transitions, we could also be in states $2, 4, 5, 6, 8$. This set of states ($= \{1, 2, 4, 5, 6, 8\}$)[3] forms a meta-state (let's call it A). From the meta-state A, where could we wind up if we read an a. Since the meta-state A includes state 2, we can follow the a to state 3. From there we can follow Λs to $2, 4, 5, 6, 8$. This gives another meta-state. Let's call it $B = \{2, 3, 4, 5, 6, 8\}$.

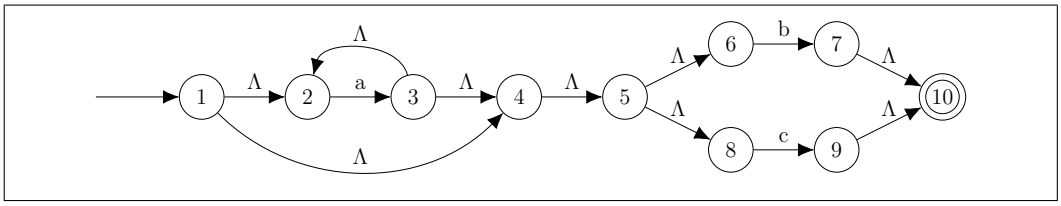


Figure 10.4: This is the NFA produced by Thompson's for the regular expression `a* (b | c)`

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 10.1.

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-

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

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 10.1: The results of performing the subset construction on the NFA in Figure 10.4. 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.

state A, and for the example NFA in Figure 10.4, the characters in the source language are a, b, c meaning three rows for each meta-state.

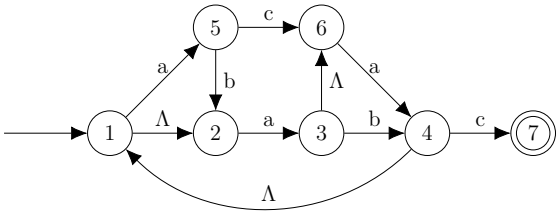
3. For each NFA state in the meta-state, if there is an outbound transition on the input, write down the destination state in the resulting states column.

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This gives us the following table:

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

All rows are filled in, so we are done.

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

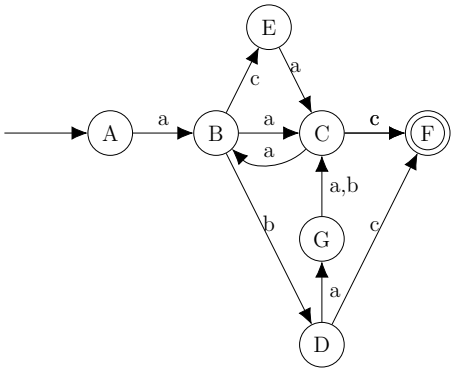
The subset construction was supposed to turn an NFA into a DFA, but it appears to have just produced a table. We can take the information in our final table from the example we just completed and summarize it in Figure 10.5.

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

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

Figure 10.5: A summary of the table derived in this section. Each row contains the start state, and input character, and the resulting state if that character is found while in the start state.

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



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

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

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

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

Chapter 11

Scanner Code

The previous chapter presented algorithms for converting an arbitrary regular express into a transition table for a deterministic finite automata. The next question is, what do we do with that table? This chapter answers that question.

11.1 Processing Transition Tables

A DFA is a state diagram, and the transition table can be used as a lookup table for performing state transitions. Before showing the code for processing the lookup table, let's make one more modification to the table presented in Figure 10.5. Instead of one row for each state-input combination, we will present the table as one row per state, and one column for each input possibility. Here is the result:

State	**a**	**b**	**c**
A	B	-	-
B	C	D	C
C	B	-	F
D	G	-	F
E	C	-	-
G	C	C	-

This table can be represented as a 2D array. The left-most column does not need to be stored as it simply gives the row index (converting from letters to suitable numbers). The current state is used as the row index and the just-read character is used as the column index. This results in the code in Listing 11.1.

```
state = 0;
while (more_input_available)
{
   input = next_character();
   state = table[state][input];
}
if (state == final_state)
   return found;
else
   return not_found;
```

Listing 11.1: Pseudo-code for processing a transition table.

This code makes some problematic simplifying assumptions. First of all, how should the code deal with non-existent transitions (all the dashes in the table)? A simply solution is to add an error state to the table. All the dashes get changed to that error state, and all the columns for that error state return back to the same state: once you're in the error state, you're stuck there.

The second problem is that the table only has three columns one each for the three input characters. How do you convert from the character read (probably a number in ASCII or Unicode). There are several possible solutions. If the compiled language is defined over ASCII characters (particularly if it is 7-bit ASCII) then the table can be extended to have a row for each possible input character.

With Unicode, this is less workable. For UTF-8 a similar scheme could be used, but the table would need additional rows to handle multi-byte characters. For UTF-16 or larger, the transition table would be so large, it would likely be unworkable. It would be possible to convert the UTF-16 to UTF-8 (they both encode the complete Unicode character set) and then use a 256 column wide table. This would cost the compiler time at the expense of a potentially smaller table.

Since this is A Very Simple compiler book, I will not delve into character conversions. Instead I'll assume the compiled language can use (or convert to) a reasonable sized character set.

11.1.1 Replacing the table with code

There are many space/speed trade-offs in computer algorithms (such as executing code to convert from UTF-16 to UTF-8 to have a narrower transition table). There are also trade-offs between code size and data size. The code presented in Listing 11.1 was very compact, but it required a potentially large data table. There is an alternative implementation that requires very little data, but a large amount of code. This is illustrated in Listing 11.2.

```
state_0:
input = next_character();
switch (input)
{
   case 'a': goto state_1;
   case 'b': goto state_2;
   case 'c': goto state_done;
   default: goto state_error;
}

state_1:
input = next_character();
switch (input)
{
   case 'x': goto state_0;
   case 'b': goto state_5;
   case EOF: goto state_done;
   default: goto state_error;
}
...
```

Listing 11.2: Alternative pseudo-code for processing a transition table. This just a sample to illustrate the form of the code. The actual `switch` statements would be determined based on the transition table.

This version of the code uses `switch` statements to move from state to state. Essentially, each row of the transition table is encoded into a `switch` statement. `goto` statements[1] are used to jump to the block of code that handles the new state.

One advantage of this implementation is that the code (and data) is not affected by large character sets. Each `switch` statement only has branches for characters that matter. Any unused characters in the character set will not appear in the code. The flip side of this is that if there are states that can handle a large number of characters (for example, the regular expression for an identifier in a UNICODE language could include any of a very large number of characters) would require a large number of `case` branches.

11.1.2 Performance considerations

So, which of these two implementations is better. The short answer is: that depends.

The first consideration is how efficiently the implementation language implements a `switch` statement. If the `switch` statements are translated to an equivalent `if elsif else` statement, then there is likely little advantage to the `switch` form of implementation. For the following discussion, we'll assume an efficient `switch` implementation.

Implementation size

If we consider the memory size required by the implementation, the following observations can be made:

1. The code for the transition table version is quite small. The size of the code does not change with the complexity of the language being scanned: the same code can process any transition table.

[1] Normally, `goto` statements are considered an unforgivable evil in code. This is one of the few instances were they are useful, and not evil

2. The data table for the transition table version can be quite large. Its size depends on the number of characters in the character set for the language being compiled, and on the number of states that are required to implement the scanner.

3. The code for the `switch` implementation can be very large. It depends on the number of states that are required to implement the scanner and also on the number of valid characters in each state.

4. The data required for the `switch` implementation is small: a single integer to store the current character.

If the transition table is sparse (few valid characters in each row), then the `switch` implementation may require less memory because the `switch` statements only include code for the necessary characters whereas the transition table version must store the entire table–even the cells for transitions that are invalid. However, if the table is dense (many valid characters in each (or many) rows, then the transition table version is likely to require less memory. Each valid entry in the table requires more code in the `switch` version, however the table size doesn't change with density in the transition table implementation.

Locality of Reference

This section needs work.

11.2 Processing multiple regular expressions

Algorithm for multiple RE's

This section needs work.

Chapter 12

Automatically Generated Scanners

here is a description of flex.

This section needs work.

Part IV

Parsing

The scanner converts the input stream from a stream of characters into a stream of tokens. The parser tries to make sense of the stream of tokens. In particular, it answers the question, "Does this sequence of tokens for a valid program?"

Chapter 13

Context Free Grammars

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

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

this symbol can be replaced by this collection of symbols

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

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

Symbols in CFGs are of two flavors: non-terminals are those that appear on the left hand side of a production. They are non-terminals because they can be replaced by other symbols. Terminals are those symbols that never appear on the left hand side. They are "terminal" because they can never be replaced. For ease of reading, non-terminals are usually given in UPPER-CASE. terminals are given in lowercase. CFGs also need a start symbol - the

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

```
1 S → a S b
2 S → λ
```

CFG 13.1: A CFG that defines the language of any number of `a`'s followed by the same number of `b`'s.

Figure 13.1 shows a complete CFG. The productions have been numbered for easy reference. There are only two productions. The first one says that the start symbol (`S`) can be replaced with `a S b`. Note that this is a recursive rule because the `S` appears on both sides. The second production says that `S` can be replaced with nothing.

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

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

While not the most complex illustration, this CFG illustrates that CFGs are more powerful than regular expressions. Regular expressions are not able to generate balanced parenthesis. A regular expression such as `(*)*` allows any number of opening parenthesis and any number of closing parenthesis, but there is no way to guarantee that the number of closing parenthesis match the number of opening parenthesis. If we substituted parenthesis for the `a` and `b` in Figure 13.1, we would have a solution to the balanced parenthesis problem.

Figure 13.1 presents a more complete, and more interesting CFG. This language defines a program as zero or more statements. An individual statement can be an assignment statement (in this language, an assignment statement is a terminal, so the assumption is that they are defined elsewhere), an `if`

statement, or a compound statement (curly braces surrounding any number of statements).

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

Figure 13.1: This CFG defines a program as being zero or more statements, where each statement is either an `if` statement, an assignment statement or a compound statement.

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

Listing 13.2: Sample program in the language defined by the CFG in Figure 13.1

The program in Listing 13.2 illustrates the features of the language defined by the CFG in Figure 13.1. Line 1 is a simple assignment statement. The `if` statement that begins in Line 2 is a simple `if` statement. The `if` statement that begins in Line 4 shows a nested `if` statement. The compound statement that begins in Line 8 shows that compound statements can be nested and that they can be empty (in Line 11).

13.1 Derivations

Listing 13.2 claims to be a program in the language defined in Figure 13.1. How can we substantiate this claim? This is normally done by showing a derivation of the program given the CFG. Each line of a derivation substitutes a single non-terminal for the right-hand-side of a production for that non-terminal.

Derivations start with the start symbol and continue until there are only non-terminals. Each step other than the first should list the production number that was invoked to make the substitution. Rather than starting with the longer program in Listing 13.2, let's start with the shorter program in Listing 13.3.

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

Listing 13.3: Short program program in the language defined by the CFG in Figure 13.1. The derivation of this program is given in Derivation 13.4

zzz

```
   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 13.4: Derivation of the program in Listing 13.3

Chapter 14

Top-down recursive-descent parsers

This section needs work.

Chapter 15

Bottom-up parsers

This section needs work.

Chapter 16

Automatically generated parsers

Description of bison.

This section needs work.

Part V

Semantic Processing

This section needs work.

Chapter 17

Syntax vs. Semantics

This section needs work.

Chapter 18

Type systems

This section needs work.

Chapter 19

Implementation details

This section needs work.

Part VI

The Back-end

This section needs work.

Chapter 20

The Visitor Pattern

This section needs work.

Chapter 21

Code generation

This section needs work.

Chapter 22

Optimization

This section needs work.

Part VII

LaTeXsample code

Description List Each description list item has a term followed by the description of that term.

Bunyip Mythical beast of Australian Aboriginal legends.

22.1 Theorem-Like Environments

The following theorem-like environments (in alphabetical order) are available in this style.

Acknowledgement 2 *This is an acknowledgement*

Algorithm 3 *This is an algorithm*

Axiom 4 *This is an axiom*

Case 5 *This is a case*

Claim 6 *This is a claim*

Conclusion 7 *This is a conclusion*

Condition 8 *This is a condition*

Conjecture 9 *This is a conjecture*

Corollary 10 *This is a corollary*

Criterion 11 *This is a criterion*

Definition 12 *This is a definition*

Example 13 *This is an example*

Exercise 14 *This is an exercise*

Lemma 15 *This is a lemma*

Proof. This is the proof of the lemma. ■

Notation 16 *This is notation*

Problem 17 *This is a problem*

Proposition 18 *This is a proposition*

Remark 19 *This is a remark*

Summary 20 *This is a summary*

Theorem 21 *This is a theorem*

Proof of the Main Theorem. This is the proof. ■

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.