It can be proven, constructively, that regular expressions and finite automata are equivalent: one can construct a finite automaton that accepts the language defined by a given regular expression, and vice versa. Similarly, it is possible to construct a push-down automaton that accepts the language defined by a given context-free grammar, and vice versa. The grammar-to-automaton constructions are in fact performed by scanner and parser generators such as `lex` and `yacc`. Of course, a real scanner does not accept just one token; it is called in a loop so that it keeps accepting tokens repeatedly. As noted in Sidebar 2.4, this detail is accommodated by having the scanner accept the alternation of all the tokens in the language (with distinguished final states), and by having it continue to consume characters until no longer token can be constructed.

IN MORE DEPTH

On the companion site we consider finite and pushdown automata in more detail. We give an algorithm to convert a DFA into an equivalent regular expression. Combined with the constructions in Section 2.2.1, this algorithm demonstrates the equivalence of regular expressions and finite automata. We also consider the sets of grammars and languages that can and cannot be parsed by the various linear-time parsing algorithms.

2.5 Summary and Concluding Remarks

In this chapter we have introduced the formalisms of regular expressions and context-free grammars, and the algorithms that underlie scanning and parsing in practical compilers. We also mentioned syntax error recovery, and presented a quick overview of relevant parts of automata theory. Regular expressions and context-free grammars are language *generators*: they specify how to construct valid strings of characters or tokens. Scanners and parsers are language *recognizers*: they indicate whether a given string is valid. The principal job of the scanner is to reduce the quantity of information that must be processed by the parser, by grouping characters together into tokens, and by removing comments and white space. Scanner and parser generators automatically translate regular expressions and context-free grammars into scanners and parsers.

Practical parsers for programming languages (parsers that run in linear time) fall into two principal groups: top-down (also called LL or predictive) and bottom-up (also called LR or shift-reduce). A top-down parser constructs a parse tree starting from the root and proceeding in a left-to-right depth-first traversal. A bottom-up parser constructs a parse tree starting from the leaves, again working left-to-right, and combining partial trees together when it recognizes the children of an internal node. The stack of a top-down parser contains a prediction of what will be seen in the future; the stack of a bottom-up parser contains a record of what has been seen in the past.

Top-down parsers tend to be simple, both in the parsing of valid strings and in the recovery from errors in invalid strings. Bottom-up parsers are more powerful, and in some cases lend themselves to more intuitively structured grammars, though they suffer from the inability to embed action routines at arbitrary points in a right-hand side (we discuss this point in more detail in Section C-4.5.1). Both varieties of parser are used in real compilers, though bottom-up parsers are more common. Top-down parsers tend to be smaller in terms of code and data size, but modern machines provide ample memory for either.

Both scanners and parsers can be built by hand if an automatic tool is not available. Handbuilt scanners are simple enough to be relatively common. Handbuilt parsers are generally limited to top-down recursive descent, and are most commonly used for comparatively simple languages. Automatic generation of the scanner and parser has the advantage of increased reliability, reduced development time, and easy modification and enhancement.

Various features of language design can have a major impact on the complexity of syntax analysis. In many cases, features that make it difficult for a compiler to scan or parse also make it difficult for a human being to write correct, maintainable code. Examples include the lexical structure of Fortran and the `if... then... else` statement of languages like Pascal. This interplay among language design, implementation, and use will be a recurring theme throughout the remainder of the book.

2.6 Exercises

2.1 Write regular expressions to capture the following.

(a) Strings in C. These are delimited by double quotes (`"`), and may not contain newline characters. They may contain double-quote or backslash characters if and only if those characters are “escaped” by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is *not* a member of a small specified set.

(b) Comments in Pascal. These are delimited by `(*` and `*)` or by `{` and `}`. They are not permitted to nest.

(c) Numeric constants in C. These are octal, decimal, or hexadecimal integers, or decimal or hexadecimal floating-point values. An octal integer begins with `0`, and may contain only the digits `0`–`7`. A hexadecimal integer begins with `0x` or `0X`, and may contain the digits `0`–`9` and `a`/`A`–`f`/`F`. A decimal floating-point value has a fractional portion (beginning with a dot) or an exponent (beginning with `E` or `e`). Unlike a decimal integer, it is allowed to start with `0`. A hexadecimal floating-point value has an optional fractional portion and a mandatory exponent (beginning with `P` or `p`). In either decimal or hexadecimal, there may be digits to the left of the dot, the right of the dot, or both, and the exponent itself is given in decimal, with an optional leading + or - sign. An integer may end with an optional `U` or `u` (indicating "unsigned"), and/or `L` or `l` (indicating "long") or `LL` or `ll` (indicating "long long"). A floating-point value may end with an optional `F` or `f` (indicating "float"—single precision) or `L` or `l` (indicating "long"—double precision).

(d) Floating-point constants in Ada. These match the definition of *real* in Example 2.3, except that (1) a digit is required on both sides of the decimal point, (2) an underscore is permitted between digits, and (3) an alternative numeric base may be specified by surrounding the nonexponent part of the number with pound signs, preceded by a base in decimal (e.g., `16#6.a7#e+2`). In this latter case, the letters `a..f` (both upper- and lowercase) are permitted as digits. Use of these letters in an inappropriate (e.g., decimal) number is an error, but need not be caught by the scanner.

(e) Inexact constants in Scheme. Scheme allows real numbers to be explicitly *inexact* (imprecise). A programmer who wants to express all constants using the same number of characters can use sharp signs (`#`) in place of any lower-significance digits whose values are not known. A base-10 constant without exponent consists of one or more digits followed by zero of more sharp signs. An optional decimal point can be placed at the beginning, the end, or anywhere in-between. (For the record, numbers in Scheme are actually a good bit more complicated than this. For the purposes of this exercise, please ignore anything you may know about sign, exponent, radix, exactness and length specifiers, and complex or rational values.)

(f) Financial quantities in American notation. These have a leading dollar sign (`$`), an optional string of asterisks (`*`—used on checks to discourage fraud), a string of decimal digits, and an optional fractional part consisting of a decimal point (`.`) and two decimal digits. The string of digits to the left of the decimal point may consist of a single zero (`0`). Otherwise it must not start with a zero. If there are more than three digits to the left of the decimal point, groups of three (counting from the right) must be separated by commas (`,`). Example: `$**2,345.67`. (Feel free to use "productions" to define abbreviations, so long as the language remains regular.)

2.2 Show (as "circles-and-arrows" diagrams) the finite automata for Exercise 2.1.

2.3 Build a regular expression that captures all nonempty sequences of letters other than `file`, `for`, and `from`. For notational convenience, you may assume the existence of a **not** operator that takes a set of letters as argument and matches any *other* letter. Comment on the practicality of constructing a regular expression for all sequences of letters other than the keywords of a large programming language.

2.4 (a) Show the NFA that results from applying the construction of Figure 2.7 to the regular expression *letter* (*letter* | *digit*)*.

(b) Apply the transformation illustrated by Example 2.14 to create an equivalent DFA.

(c) Apply the transformation illustrated by Example 2.15 to minimize the DFA.

2.5 Starting with the regular expressions for *integer* and *decimal* in Example 2.3, construct an equivalent NFA, the set-of-subsets DFA, and the minimal equivalent DFA. Be sure to keep separate the final states for the two different kinds of token (see Sidebar 2.4). You may find the exercise easier if you undertake it by modifying the machines in Examples 2.13 through 2.15.

2.6 Build an ad hoc scanner for the calculator language. As output, have it print a list, in order, of the input tokens. For simplicity, feel free to simply halt in the event of a lexical error.

2.7 Write a program in your favorite scripting language to remove comments from programs in the calculator language (Example 2.9).

2.8 Build a nested-`case`-statements finite automaton that converts all letters in its input to lower case, except within Pascal-style comments and strings. A Pascal comment is delimited by `{` and `}`, or by `(*` and `*)`. Comments do not nest. A Pascal string is delimited by single quotes (`' ... '`). A quote character can be placed in a string by doubling it (`'Madam, I''m Adam.'`). This upper-to-lower mapping can be useful if feeding a program written in standard Pascal (which ignores case) to a compiler that considers upper- and lowercase letters to be distinct.

2.9 (a) Describe in English the language defined by the regular expression `a* ( b a* b a* )*`. Your description should be a high-level characterization—one that would still make sense if we were using a different regular expression for the same language.

(b) Write an unambiguous context-free grammar that generates the same language.

(c) Using your grammar from part (b), give a canonical (right-most) derivation of the string `b a a b a a a b b`.

2.10 Give an example of a grammar that captures right associativity for an exponentiation operator (e.g., `**` in Fortran).

2.11 Prove that the following grammar is LL(1):

decl ⟶ `ID` *decl_tail*
decl_tail ⟶ `,` *decl*
⟶ `:` `ID` `;`

(The final `ID` is meant to be a type name.)

2.12 Consider the following grammar:

$$\begin{array}{lcl}
G & \longrightarrow & S \ \$\$ \\
S & \longrightarrow & A\ M \\
M & \longrightarrow & S \mid \epsilon \\
A & \longrightarrow & \texttt{a}\ E \mid \texttt{b}\ A\ A \\
E & \longrightarrow & \texttt{a}\ B \mid \texttt{b}\ A \mid \epsilon \\
B & \longrightarrow & \texttt{b}\ E \mid \texttt{a}\ B\ B
\end{array}$$

(a) Describe in English the language that the grammar generates.
(b) Show a parse tree for the string `a b a a`.
(c) Is the grammar LL(1)? If so, show the parse table; if not, identify a prediction conflict.

2.13 Consider the following grammar:

$$\begin{array}{lcl}
\textit{stmt} & \longrightarrow & \textit{assignment} \\
 & \longrightarrow & \textit{subr\_call} \\
\textit{assignment} & \longrightarrow & \texttt{id}\ \texttt{:=}\ \textit{expr} \\
\textit{subr\_call} & \longrightarrow & \texttt{id}\ \texttt{(}\ \textit{arg\_list}\ \texttt{)} \\
\textit{expr} & \longrightarrow & \textit{primary}\ \textit{expr\_tail} \\
\textit{expr\_tail} & \longrightarrow & \textit{op}\ \textit{expr} \\
 & \longrightarrow & \epsilon \\
\textit{primary} & \longrightarrow & \texttt{id} \\
 & \longrightarrow & \textit{subr\_call} \\
 & \longrightarrow & \texttt{(}\ \textit{expr}\ \texttt{)} \\
\textit{op} & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\textit{arg\_list} & \longrightarrow & \textit{expr}\ \textit{args\_tail} \\
\textit{args\_tail} & \longrightarrow & \texttt{,}\ \textit{arg\_list} \\
 & \longrightarrow & \epsilon
\end{array}$$

(a) Construct a parse tree for the input string `foo(a, b)`.
(b) Give a canonical (right-most) derivation of this same string.
(c) Prove that the grammar is not LL(1).
(d) Modify the grammar so that it *is* LL(1).

2.14 Consider the language consisting of all strings of properly balanced parentheses and brackets.

(a) Give LL(1) and SLR(1) grammars for this language.
(b) Give the corresponding LL(1) and SLR(1) parsing tables.
(c) For each grammar, show the parse tree for `([]([]))[](())`.
(d) Give a trace of the actions of the parsers in constructing these trees.

2.15 Consider the following context-free grammar.

$$\begin{array}{lcl}
G & \longrightarrow & G\ B \\
 & \longrightarrow & G\ N \\
 & \longrightarrow & \epsilon \\
B & \longrightarrow & \texttt{(}\ E\ \texttt{)} \\
E & \longrightarrow & E\ \texttt{(}\ E\ \texttt{)} \\
 & \longrightarrow & \epsilon \\
N & \longrightarrow & \texttt{(}\ L\ \texttt{]} \\
L & \longrightarrow & L\ E \\
 & \longrightarrow & L\ \texttt{(} \\
 & \longrightarrow & \epsilon
\end{array}$$

(a) Describe, in English, the language generated by this grammar. (Hint: B stands for "balanced"; N stands for "nonbalanced".) (Your description should be a high-level characterization of the language—one that is independent of the particular grammar chosen.)
(b) Give a parse tree for the string `((]()`.
(c) Give a canonical (right-most) derivation of this same string.
(d) What is FIRST(E) in our grammar? What is FOLLOW(E)? (Recall that FIRST and FOLLOW sets are defined for symbols in an arbitrary CFG, regardless of parsing algorithm.)
(e) Given its use of left recursion, our grammar is clearly not LL(1). Does this language have an LL(1) grammar? Explain.

2.16 Give a grammar that captures all levels of precedence for arithmetic expressions in C, as shown in Figure 6.1. (Hint: This exercise is somewhat tedious. You'll probably want to attack it with a text editor rather than a pencil.)

2.17 Extend the grammar of Figure 2.25 to include `if` statements and `while` loops, along the lines suggested by the following examples:

```
abs := n
if n < 0 then abs := 0 - abs fi

sum := 0
read count
while count > 0 do
    read n
    sum := sum + n
    count := count - 1
od
write sum
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an `if` or `while` statement.

2.18 Consider the following LL(1) grammar for a simplified subset of Lisp:

```
P  ⟶ E $$
E  ⟶ atom
   ⟶ ' E
   ⟶ ( E Es )
Es ⟶ E Es
   ⟶
```

(a) What is FIRST(*Es*)? FOLLOW(*E*)? PREDICT(*Es* ⟶ ϵ)?

(b) Give a parse tree for the string `(cdr '(a b c)) $$`.

(c) Show the left-most derivation of `(cdr '(a b c)) $$`.

(d) Show a trace, in the style of Figure 2.21, of a table-driven top-down parse of this same input.

(e) Now consider a recursive descent parser running on the same input. At the point where the quote token (`'`) is matched, which recursive descent routines will be active (i.e., what routines will have a frame on the parser's run-time stack)?

2.19 Write top-down and bottom-up grammars for the language consisting of all well-formed regular expressions. Arrange for all operators to be left-associative. Give Kleene closure the highest precedence and alternation the lowest precedence.

2.20 Suppose that the expression grammar in Example 2.8 were to be used in conjunction with a scanner that did *not* remove comments from the input, but rather returned them as tokens. How would the grammar need to be modified to allow comments to appear at arbitrary places in the input?

2.21 Build a complete recursive descent parser for the calculator language. As output, have it print a trace of its matches and predictions.

2.22 Extend your solution to Exercise 2.21 to build an explicit parse tree.

2.23 Extend your solution to Exercise 2.21 to build an abstract syntax tree directly, without constructing a parse tree first.

2.24 The dangling `else` problem of Pascal was not shared by its predecessor Algol 60. To avoid ambiguity regarding which `then` is matched by an `else`, Algol 60 prohibited `if` statements immediately inside a `then` clause. The Pascal fragment

```
if C1 then if C2 then S1 else S2
```

had to be written as either

```
if C1 then begin if C2 then S1 end else S2
```

or

```
if C1 then begin if C2 then S1 else S2 end
```

in Algol 60. Show how to write a grammar for conditional statements that enforces this rule. (Hint: You will want to distinguish in your grammar between conditional statements and nonconditional statements; some contexts will accept either, some only the latter.)

2.25 Flesh out the details of an algorithm to eliminate left recursion and common prefixes in an arbitrary context-free grammar.

2.26 In some languages an assignment can appear in any context in which an expression is expected: the value of the expression is the right-hand side of the assignment, which is placed into the left-hand side as a side effect. Consider the following grammar fragment for such a language. Explain why it is not LL(1), and discuss what might be done to make it so.

```
expr        ⟶ id := expr
            ⟶ term term_tail
term_tail   ⟶ + term term_tail | ε
term        ⟶ factor factor_tail
factor_tail ⟶ * factor factor_tail | ε
factor      ⟶ ( expr ) | id
```

2.27 Construct the CFSM for the *id_list* grammar in Example 2.20 and verify that it can be parsed bottom-up with *zero* tokens of look-ahead.

2.28 Modify the grammar in Exercise 2.27 to allow an *id_list* to be empty. Is the grammar still LR(0)?

2.29 Repeat Example 2.36 using the grammar of Figure 2.15.

2.30 Consider the following grammar for a declaration list:

```
decl_list ⟶ decl_list decl ; | decl ;
decl      ⟶ id : type
type      ⟶ int | real | char
          ⟶ array const .. const of type
          ⟶ record decl_list end
```

Construct the CFSM for this grammar. Use it to trace out a parse (as in Figure 2.30) for the following input program:

```
foo : record
        a : char;
        b : array 1 .. 2 of real;
    end;
```

2.31–2.37 In More Depth.