2.2 Scanning

Together, the scanner and parser for a programming language are responsible for discovering the syntactic structure of a program. This process of discovery, or *syntax analysis*, is a necessary first step toward translating the program into an equivalent program in the target language. (It's also the first step toward interpreting the program directly. In general, we will focus on compilation, rather than interpretation, for the remainder of the book. Most of what we shall discuss either has an obvious application to interpretation, or is obviously irrelevant to it.)

By grouping input characters into tokens, the scanner dramatically reduces the number of individual items that must be inspected by the more computationally intensive parser. In addition, the scanner typically removes comments (so the parser doesn't have to worry about them appearing throughout the context-free grammar—see Exercise 2.20); saves the text of "interesting" tokens like identifiers, strings, and numeric literals; and tags tokens with line and column numbers, to make it easier to generate high-quality error messages in subsequent phases.

EXAMPLE 2.9

Tokens for a calculator language

In Examples 2.4 and 2.8 we considered a simple language for arithmetic expressions. In Section 2.3.1 we will extend this to create a simple "calculator language" with input, output, variables, and assignment. For this language we will use the following set of tokens:

assign ⟶ `:=`
plus ⟶ `+`
minus ⟶ `-`
times ⟶ `*`
div ⟶ `/`
lparen ⟶ `(`
rparen ⟶ `)`
id ⟶ *letter* (*letter* | *digit*)*
except for `read` and `write`
number ⟶ *digit* *digit* * | *digit* * (. *digit* | *digit* .) *digit* *

In keeping with Algol and its descendants (and in contrast to the C-family languages), we have used `:=` rather than `=` for assignment. For simplicity, we have omitted the exponential notation found in Example 2.3. We have also listed the tokens `read` and `write` as exceptions to the rule for `id` (more on this in Section 2.2.2). To make the task of the scanner a little more realistic, we borrow the two styles of comment from C:

comment ⟶ `/*` (*non-*`*` | `*` *non-*`/`)* `*`+ `/`
| `//` (*non-newline*)* *newline*

Here we have used *non-*`*`, *non-*`/`, and *non-newline* as shorthand for the alternation of all characters other than `*`, `/`, and *newline*, respectively. ■

EXAMPLE 2.10

An ad hoc scanner for calculator tokens

How might we go about recognizing the tokens of our calculator language? The simplest approach is entirely ad hoc. Pseudocode appears in Figure 2.5. We can structure the code however we like, but it seems reasonable to check the simpler and more common cases first, to peek ahead when we need to, and to embed loops for comments and for long tokens such as identifiers and numbers.

After finding a token the scanner returns to the parser. When invoked again it repeats the algorithm from the beginning, using the next available characters of input (including any that were peeked at but not consumed the last time). ■

As a rule, we accept the longest possible token in each invocation of the scanner. Thus `foobar` is always `foobar` and never `f` or `foo` or `foob`. More to the point, in a language like C, `3.14159` is a real number and never `3`, `.`, and `14159`. White space (blanks, tabs, newlines, comments) is generally ignored, except to the extent that it separates tokens (e.g., `foo bar` is different from `foobar`).

Figure 2.5 could be extended fairly easily to outline a scanner for some larger programming language. The result could then be fleshed out, by hand, to create code in some implementation language. Production compilers often use such ad hoc scanners; the code is fast and compact. During language development, however, it is usually preferable to build a scanner in a more structured way, as an explicit representation of a *finite automaton*. Finite automata can be generated automatically from a set of regular expressions, making it easy to regenerate a scanner when token definitions change.

EXAMPLE 2.11

Finite automaton for a calculator scanner

An automaton for the tokens of our calculator language appears in pictorial form in Figure 2.6. The automaton starts in a distinguished initial state. It then moves from state to state based on the next available character of input. When it reaches one of a designated set of final states it recognizes the token associated with that state. The "longest possible token" rule means that the scanner returns to the parser only when the next character cannot be used to continue the current token. ■

DESIGN & IMPLEMENTATION

2.3 Nested comments

Nested comments can be handy for the programmer (e.g., for temporarily "commenting out" large blocks of code). Scanners normally deal only with nonrecursive constructs, however, so nested comments require special treatment. Some languages disallow them. Others require the language implementor to augment the scanner with special-purpose comment-handling code. C and C++ strike a compromise: `/* ... */` style comments are not allowed to nest, but `/* ... */` and `// ...` style comments can appear inside each other. The programmer can thus use one style for "normal" comments and the other for "commenting out." (The C99 designers note, however, that conditional compilation (`#if`) is preferable [Int03a, p. 58].)

```
skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '-', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return assign else announce an error
if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or newline is seen, respectively
        jump back to top of code
    else return div
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

Figure 2.5 Outline of an ad hoc scanner for tokens in our calculator language.

2.2.1 Generating a Finite Automaton

While a finite automaton can in principle be written by hand, it is more common to build one automatically from a set of regular expressions, using a *scanner generator* tool. For our calculator language, we should like to covert the regular expressions of Example 2.9 into the automaton of Figure 2.6. That automaton has the desirable property that its actions are *deterministic*: in any given state with a given input character there is never more than one possible outgoing transition (arrow) labeled by that character. As it turns out, however, there is no obvious one-step algorithm to convert a set of regular expressions into an equivalent deterministic finite automaton (DFA). The typical scanner generator implements the conversion as a series of three separate steps.

The first step converts the regular expressions into a *nondeterministic* finite automaton (NFA). An NFA is like a DFA except that (1) there may be more than one transition out of a given state labeled by a given character, and (2) there may be so-called *epsilon transitions*: arrows labeled by the empty string symbol, ϵ. The NFA is said to accept an input string (token) if there exists a path from the start