Chapter 3 – Scanning

3.1 Kinds of Tokens

Scanning is the process of identifying tokens from the raw text source code of a program. At first glance, scanning might seem trivial – after all, identifying words in a natural language is as simple as looking for spaces between letters. However, identifying tokens in source code requires the language designer to clarify many fine details, so that it is clear what is permitted and what is not.

Most languages will have tokens in these categories:

- **Keywords** are words in the language structure itself, like `while` or `class` or `true`. Keywords must be chosen carefully to reflect the natural structure of the language, without interfering with the likely names of variables and other identifiers.

- **Identifiers** are the names of variables, functions, classes, and other code elements chosen by the programmer. Typically, identifiers are arbitrary sequences of letters and possibly numbers. Some languages require identifiers to be marked with a sentinel (like the dollar sign in Perl) to clearly distinguish identifiers from keywords.

- **Numbers** could be formatted as integers, or floating point values, or fractions, or in alternate bases such as binary, octal or hexadecimal. Each format should be clearly distinguished, so that the programmer does not confuse one with the other.

- **Strings** are literal character sequences that must be clearly distinguished from keywords or identifiers. Strings are typically quoted with single or double quotes, but also muts have some facility for containing quotations, newlines, and unprintable characters.

- **Comments** and **whitespace** are used to format a program to make it visually clear, and in some cases (like Python) are significant to the structure of a program.

When designing a new language, or designing a compiler for an existing language, the first job is to state precisely what characters are permitted in each type of token. Initially, this could be done informally by stating,
token_t scan_token( FILE *fp ) {
    char c = fgetc(fp);
    if(c=='*') {
        return TOKEN_MULTIPLY;
    } else if(c=='!') {
        char d = fgetc(fp);
        if(d=='=') {
            return TOKEN_NOT_EQUAL;
        } else {
            ungetc(d,fp);
            return TOKEN_NOT;
        }
    } else if(isalpha(c)) {
        do {
            char d = fgetc(fp);
        } while(isalphanum(d));
        ungetc(d,stdin);
        return TOKEN_IDENTIFIER;
    } else if ( . . . ) {
        . . .
    }
}

Figure 3.1: A Simple Hand Made Scanner

for example, “An identifier consists of a letter followed by any number of letters and numerals,”, and then assigning a symbolic constant (TOKEN_IDENTIFIER for that kind of token. As we will see, an informal approach is often ambiguous, and a more rigorous approach is needed.

3.2 A Hand-Made Scanner

Figure 3.1 shows how one might write a scanner by hand, using simple coding techniques. To keep things simple, we only consider just a few tokens: * for multiplication, ! for logical-not, != for not-equal, and sequences of letters and numbers for identifiers.

The basic approach is to read one character at a time from the input stream (fgetc(fp)) and then classify it. Some single-character tokens are easy: if the scanner reads a * character, it immediately returns TOKEN_MULTIPLY, and the same would be true for addition, subtraction, and so forth.

However, some characters are part of multiple tokens. If the scanner encounters !, that could represent a logical-not operation by itself, or it could be the first character in the != sequence representing not-equal-to. Upon reading !, the scanner must immediately read the next character. If
the next character is =, then it has matched the sequence != and returns
TOKEN_NOT_EQUAL.

But, if the character following ! is something else, then the non-matching
character needs to be put back on the input stream using ungetc, because
it is not part of the current token. The scanner returns TOKEN_NOT and will
consume the put-back character on the next call to scan_token.

In a similar way, once a letter has been identified by isalpha(c), then
the scanner keeps reading letters or numbers, until a non-matching char-
acter is found. The non-matching character is put back, and the scanner
returns TOKEN_IDENTIFIER.

(We will see this pattern come up in every stage of the compiler: an
unexpected item doesn’t match the current objective, so it must be put
back for later. This is known more generally as backtracking.)

As you can see, a hand-made scanner is rather verbose. As more to-
ken types are added, the code can become quite convoluted, particularly
if tokens share common sequences of characters. It can also be difficult
for a developer to be certain that the scanner code corresponds to the de-
sired definition of each token, which can result in unexpected behavior on
complex inputs. That said, for a small language with a limited number of
tokens, a hand-made scanner can be an appropriate solution.

For a complex language with a large number of tokens, we need a more
formalized approach to defining and scanning tokens. A formal approach
will allow us to have a greater confidence that token definitions do not
conflict and the scanner is implemented correctly. Further, a formalized
approach will allow us to make the scanner compact and high perfor-
ma nce – surprisingly, the scanner itself can be the performance bottleneck
in a compiler, since every single character must be individually consid-
ered.

The formal tools of regular expressions and finite automata, allow us
to state very precisely what may appear in a given token type. Then, auto-
mated tools can process these definitions, find errors or ambiguities, and
produce compact, high performance code.

3.3 Regular Expressions

Regular expressions (REs) are a language for expressing patterns. They
were first described in the 1950s by Stephen Kleene as an element of his
foundational work in automata theory and computability. Today, REs
are found in slightly different forms in programming languages (Perl),
standard libraries (PCRE), text editors (vi), command-line tools (grep),
and many other places. We can use regular expressions as a compact
and formal way of specifying the tokens accepted by the scanner of a
compiler, and then automatically translate those expressions into working
code. While easily explained, REs can be a bit tricky to use, and require
some practice in order to achieve the desired results.

DRAFT September 12, 2016
Let us define regular expressions precisely:

A **regular expression** $s$ is a string which denotes $L(s)$, a set of strings drawn from an alphabet $\Sigma$. $L(s)$ is known as the “language of $s$.”

$L(s)$ is defined inductively with the following base cases:

- If $a \in \Sigma$ then $a$ is a regular expression and $L(a) = a$.
- $\epsilon$ is a regular expression and $L(\epsilon)$ contains only the empty string.

Then, for any regular expressions $s$ and $t$:

1. $s \mid t$ is a RE such that $L(s \mid t) = L(s) \cup L(t)$.
2. $st$ is a RE such that $L(st) = L(s)$ followed by $L(t)$.
3. $s^*$ is a RE such that $L(s^*) = L(s)$ concatenated zero or more times.

Rule #3 is known as the **Kleene closure** and has the highest precedence. Rule #2 is known as **concatenation**. Rule #1 has the lowest precedence and is known as **alternation**. Parentheses can be added to adjust the order of operations in the usual way.

Here are a few examples using just the basic rules. (Note that a finite RE can indicate an infinite set.)

<table>
<thead>
<tr>
<th>Regular Expression $s$</th>
<th>Language $L(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>${ \text{hello} }$</td>
</tr>
<tr>
<td>d(o</td>
<td>i)g</td>
</tr>
<tr>
<td>moo*</td>
<td>${ \text{mo, moo, moo, ...} }$</td>
</tr>
<tr>
<td>(moo)*</td>
<td>${ \epsilon, \text{moo, moomoo, moomoomoo, ...} }$</td>
</tr>
<tr>
<td>a(b</td>
<td>a)*a</td>
</tr>
</tbody>
</table>

The syntax described on the previous page is entirely sufficient to write any regular expression. But, is it also handy to have a few helper operations built on top of the basic syntax:

- $s?$ indicates that $s$ is optional.
- $s?$ can be written as $(s \mid \epsilon)$
- $s+$ indicates that $s$ is repeated one or more times.
- $s+$ can be written as $ss^*$
- $[a-z]$ indicates any character in that range.
- $[a-z]$ can be written as $(a|b|\ldots|z)$
- $[^x]$ indicates any character except one.
- $[^x]$ can be written as $\Sigma - x$
Regular expressions also obey several algebraic properties, which make it possible to re-arrange them as needed for efficiency or clarity:

<table>
<thead>
<tr>
<th>Property</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Associativity</strong></td>
<td>$a</td>
</tr>
<tr>
<td><strong>Commutativity</strong></td>
<td>$a</td>
</tr>
<tr>
<td><strong>Distribution</strong></td>
<td>$a (b</td>
</tr>
<tr>
<td><strong>Idempotency</strong></td>
<td>$a** = a*$</td>
</tr>
</tbody>
</table>

Using regular expressions, we can precisely state what is permitted in a given token. Suppose we have a hypothetical programming language with the following informal definitions and regular expressions. For each token type, we show examples of strings that match (and do not match) the regular expression.

<table>
<thead>
<tr>
<th>Informal definition</th>
<th>Regular expression</th>
<th>Matches strings</th>
<th>Does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>An identifier is a sequence of capital letters and numbers, but a number must not</td>
<td>$[A-Z]+([A-Z]</td>
<td>[0-9])*$</td>
<td>PRINT, MODE5</td>
</tr>
<tr>
<td>come first.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A number is a sequence of digits with an optional decimal point. For clarity, the</td>
<td>$[0-9]+(.[0-9]+)?$</td>
<td>123, 3.14</td>
<td>.15, 30.</td>
</tr>
<tr>
<td>decimal point must have digits on both left and right sides.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A comment is any text (except a right angle bracket) surrounded by angle brackets.</td>
<td>$&lt;[^&gt;]*&gt;$</td>
<td>trickyster part, look left</td>
<td>this is an illegal comment</td>
</tr>
</tbody>
</table>

### 3.4 Finite Automata

A finite automaton (FA) is an abstract machine that can be used to certain forms of computation. Graphically, an FA consists of a number of states (represented by numbered circles) and a number of edges (represented by labelled arrows) between those states. Each edge is labelled with one or more symbols drawn from an alphabet $\Sigma$.

The machine begins in a start state $S_0$. For each input symbol presented to the FA, it moves to the state indicated by the edge with the same label.
as the input symbol. Some states of the FA are known as **accepting states** and are indicated by a double circle. If the FA is in an accepting state after all input is consumed, then we say the the FA **accepts** the input. We say that the FA **rejects** the input string if it ends in a non-accepting state, or if there is no edge corresponding to the current input symbol.

Every RE can be written as an FA, and vice versa. For a simple regular expression, one can construct an FA by hand. For example, here is an FA for the keyword **for**:

![FA for keyword for](image)

Here is an FA for identifiers of the form `[a-z][a-z0-9]*`

![FA for identifiers](image)

And here is an FA for numbers of the form `([1-9][0-9]*)|0`

![FA for numbers](image)

### 3.4.1 Deterministic Finite Automata

Each of these three examples is a **deterministic finite automaton** (DFA). A DFA is a special case of an FA where every state has no more than one outgoing edge for a given symbol. Put another way, a DFA has no ambiguity: for every combination of state and input symbol, there is exactly one choice of what to do next.

Because of this property, a DFA is very easy to implement in software or hardware. One integer \( c \) is needed to keep track of the current state. The transitions between states are represented by a matrix \( M[s, i] \) which encodes the next state, given the current state and input symbol. (If the
transition is not allowed, we mark it with $E$ to indicate an error.) For each symbol, we compute $c = M[s, i]$ until all the input is consumed, or an error state is reached.

### 3.4.2 Nondeterministic Finite Automata

The alternative to a DFA is a **nondeterministic finite automaton (NFA)**. An NFA is a perfectly valid FA, but it has an ambiguity that makes it somewhat more difficult to work with.

Consider the regular expression $[a-z]*ing$, which represents all lowercase words ending in the suffix *ing*. It can be represented with the following automaton:

![Automaton Diagram]

Now consider how this automaton would consume the word *sing*. It could proceed in two different ways. Once would be to move to state 0 on $s$, state 1 on $i$, state 2 on $n$, and state 3 on $g$. But the other, equally valid way would be to stay in state 0 the whole time, matching each letter to the $[a-z]$ transition. Both ways obey the transition rules, but one results in acceptance, while the other results in rejection.

The problem here is that state 0 allows for two different transitions on the symbol $i$. One is to stay in state 0 matching $[a-z]$ and the other is to move to state 1 matching $i$.

Moreover, there is no simple rule by which we can pick one path or another. If the input is *sing*, the right solution is to proceed immediately from state zero to state one on $i$. But if the input is *singing*, then we should state in state zero for the first *ing* and proceed to state one for the second *ing*.

An NFA can also have an $\epsilon$ (epsilon) transition, which represents the empty string. This transition can be taking without consuming any input symbols at all. For example, we could represent the regular expression $a*(ab|ac)$ with this NFA:
3.4. FINITE AUTOMATA  

This particular NFA presents a variety of ambiguous choices. From state zero, it could consume \( a \) and stay in state zero. Or, it could take an \( \epsilon \) to state one or state four, and then consume an \( a \) either way.

There are two common ways to interpret this ambiguity:

- **The crystal ball interpretation** suggests that the NFA somehow “knows” what the best choice is, by some means external to the NFA itself. In the example above, the NFA would choose whether to proceed to state zero, one, or two before consuming the first character, and it would always make the right choice. Needless to say, this isn’t possible in a real implementation.

- **The many-worlds interpretation** suggests that that NFA exists in all allowable states *simultaneously*. When the input is complete, if any of those states are accepting states, then the NFA has accepted the input. This interpretation is more useful for constructing a working NFA, or converting it to a DFA.

Let us use the many-worlds interpretation on the example above. Suppose that the input string is \( aaac \). Initially the NFA is in state zero. Without consuming any input, it could take an epsilon transition to states one or four. So, we can consider its initial state to be all of those statuses simultaneously. Continuing on, the NFA would traverse these states until accepting the complete string \( aaac \):

<table>
<thead>
<tr>
<th>States</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 4</td>
<td>consume a</td>
</tr>
<tr>
<td>0, 1, 2, 4, 5</td>
<td>consume a</td>
</tr>
<tr>
<td>0, 1, 2, 4, 5</td>
<td>consume a</td>
</tr>
<tr>
<td>0, 1, 2, 4, 5</td>
<td>consume c</td>
</tr>
<tr>
<td>6</td>
<td>accept</td>
</tr>
</tbody>
</table>

In principle, one can implement an NFA in software or hardware by simply keeping track of all of the possible states. But this is inefficient. In the worst case, we would need to evaluate all states for all characters on each input transition. A better approach is to convert the NFA into an equivalent DFA, as we show below.
3.5 Conversion Algorithms

Regular expressions and finite automata are all equally powerful. For every RE, there is an FA, and vice versa. However, a DFA is by far the most straightforward of the three to implement in software. In this section, we will show how to convert an RE into an NFA, then an NFA into a DFA, and then to optimize the size of the DFA.

![Figure 3.2: Relationship Between REs, NFAs, and DFAs](image)

### 3.5.1 Converting REs to NFAs

To convert a regular expression to a nondeterministic finite automata, we can follow an algorithm given first by McNaughton and Yamada [?], and then by Ken Thompson [?].

We follow the same inductive definition of regular expression as given earlier. First, we define automata corresponding to the base cases of REs:

The NFA for any character $a$ is: The NFA for an $\epsilon$ transition is:

Now, suppose that we have already constructed NFAs for the regular expressions $A$ and $B$, indicated below by rectangles. Both $A$ and $B$ have a single start state (on the left) and accepting state (on the right). If we write the concatenation of $A$ and $B$ as $AB$, then the corresponding NFA is simply $A$ and $B$ connected by an $\epsilon$ transition. The start state of $A$ becomes the start state of the combination, and the accepting state of $B$ becomes the accepting state of the combination:

The NFA for the concatenation $AB$ is:
In a similar fashion, the alternation of $A$ and $B$ written as $A \mid B$ can be expressed as two automata joined by common starting and accepting nodes, all connected by $\epsilon$ transitions:

The NFA for the alternation $A \mid B$ is:

Finally, the Kleene closure $A^*$ is constructed by taking the automaton for $A$, adding starting and accepting nodes, then adding $\epsilon$ transitions to allow zero or more repetitions:

The NFA for the Kleene closure $A^*$ is:
3.5.2 Converting NFAs to DFAs

As noted above, it is possible, but unwieldy to execute an NFA directly. Instead, we can convert any NFA into an equivalent DFA using the technique of subset construction. The basic idea is to create a DFA such that each state in the DFA corresponds to multiple states in the NFA, according to the “many-worlds” interpretation.

Suppose that we begin with an NFA consisting of states $N$ and start state $N_0$. We wish to construct an equivalent DFA consisting of states $D$ and start state $D_0$. Each $D$ state will correspond to multiple $N$ states. First, we define a helper function known as the epsilon closure:

**Epsilon closure.**
\[ \epsilon\text{-closure}(n) \text{ is the set of NFA states reachable from NFA state } n \text{ by zero or more } \epsilon \text{ transitions.} \]

Now we define the subset construction algorithm. First, we create a start state $D_0$ corresponding to the $\epsilon\text{-closure}(N_0)$. Then, for each outgoing character $c$ from the states in $D_0$, we create a new state containing the epsilon closure of the states reachable by $c$. More precisely:

**Subset Construction Algorithm.**
Given an NFA with states $N$ and start state $N_0$, create an equivalent DFA with states $D$ and start state $D_0$.

Let $D_0 = \epsilon\text{-closure}(N_0)$.
Add $D_0$ to a list.
While items remain on the list:
- Let $d$ be the next DFA state removed from the list.
- For each character $c$ in $\Sigma$:
  - Let $T$ contain all NFA states $N_k$ such that:
  - $N_j \in d$ and $N_j \xrightarrow{c} N_k$
  - Create new DFA state $D_i = \epsilon\text{-closure}(T)$
  - Add $D_i$ to end of the list.

**Figure 3.3:** Subset Construction Algorithm

Let’s work out the algorithm on the example NFA shown in Figure 3.4.

1. Compute $D_0$ which is the $\epsilon\text{-closure}(N_0)$. Since $N_1$ is reachable by $\epsilon$ from $N_0$, we can see that $D_0 = \{N_0, N_1\}$. Add $D_0$ to the work list.

2. Remove $D_0$ from the work list. The character $a$ is an outgoing transition from both $N_0$ and $N_1$. We can see that $N_0 \xrightarrow{a} N_2$ and $N_1 \xrightarrow{a} \{N_3, N_4\}$, so new state $D_1 = \{N_2, N_3, N_4\}$. Add $D_1$ to the work list.
3. Remove $D_1$ from the work list. Both $a$ and $b$ are outgoing transitions from $N_2, N_3, N_4$. We can see that $N_2 \xrightarrow{b} N_3$ and $N_3 \xrightarrow{b} N_4$, so we create a new state $D_2 = \{N_3, N_4\}$ and add it to the work list. Also, $N_3 \xrightarrow{a} N_4$, so we create a new state $D_3 = \{N_4\}$ and add that to the work list.

4. Remove $D_2$ from the work list. Both $a$ and $b$ are outgoing transitions from $N_3, N_4$ via $N_3 \xrightarrow{a,b} N_4$. We already have a state $D_3$ that contains exactly $\{N_4\}$, so there is no need to create a new state, but simply add an $a, b$ transition between $D_2$ and $D_3$.

5. Remove $D_3$ from the work list. It has no outgoing transitions, so there is nothing to do.

6. The work list is empty, so we are done.

3.5.3 Minimizing DFAs with Hopcroft’s Algorithm

The subset construction algorithm will definitely generate a valid DFA, but the DFA may possibly be very large (especially if we began with a complex NFA generated from an RE.) A large DFA will have a large transition matrix that will consume a lot of memory. If it doesn’t fit in L1 cache, the scanner could run very slowly. To address this problem, we can apply Hopcroft’s algorithm to shrink a DFA into a smaller (but equivalent) DFA.

The general approach of the algorithm is to optimistically group together all possibly-equivalent states $S$ into super-states $T$. Initially, we place all non-accepting $S$ states into super-state $T_0$ and accepting states
CHAPTER 3. SCANNING 3.6. USING A SCANNER GENERATOR

**DFA Minimization Algorithm.**

Given a DFA with states $S$, create an equivalent DFA with an equal or fewer number of states $T$.

First partition $S$ into $T$ such that:
- $T_0 = \text{non-accepting states of } S$.
- $T_1 = \text{accepting states of } S$.

Repeat:
- $\forall T_i \in T$:
  - $\forall c \in \Sigma$:
    - if $T_i \xrightarrow{c} \{ \text{more than one } T \text{ state } \}$,
      - then split $T_i$ into multiple $T$ states
        such that $c$ has the same action in each.

Until no more states are split.

Figure 3.5: Hopcroft’s DFA Minimization Algorithm

into super-state $T_1$. Then, we examine the outgoing edges in each state $s \in T_i$. If, a given character $c$ has edges that begin in $T_i$ and end if different super-states, then we consider the super-state to be inconsistent with respect to $c$. (Consider an impermissible transition as if it were a transition to $T_E$, a super-state for errors.) The super-state must then be split into multiple states that are consistent with respect to $c$. Repeat this process for all super-states and all characters $c \in \Sigma$ until no more splits are required.

### 3.6 Using a Scanner Generator

Because a regular expression precisely describes all the allowable forms of a token, we can use a program to automatically transform a set of regular expressions into code for a scanner. Such a program is known as a scanner generator. The program Lex, developed at AT&T was one of the earliest examples of a scanner generator. Flex isthe GNU replacement of Lex and is widely used in Unix-like operating systems today to generate scanners implemented in C or C++.

To use Flex, we write a specification of the scanner that is a mixture of regular expressions, fragments of C code, and some specialized directives. The Flex program itself consumes the specification and produces regular C code that can then be compiled in the normal way.

Here is the overall structure of a Flex file:

The first section consists of arbitrary C code that will be placed at the beginning of `scanner.c`, like include files, type definitions, and similar things. Typically, this is used to include a file that contains the symbolic constants for tokens.

The second section states character classes, which are a symbolic short-
hand for commonly used regular expressions. For example, you might declare \texttt{DIGIT \[0-9\]}\]. This class can be referred to later as \{DIGIT\}.

The third section is the most important part. It states a regular expression for each type of token that you wish to match, followed by a fragment of C code that will be executed whenever the expression is matched. In the simplest case, this code returns the type of the token, but it can also be used to extract token values, display errors, or anything else appropriate.

The fourth section is arbitrary C code that will go at the end of the scanner, typically for additional helper functions. A peculiar requirement of Flex is that we must define a function \texttt{yywrap} which returns zero to indicate that the input is complete at the end of the file.

The regular expression language accepted by Flex is very similar to that of formal regular expressions discussed above. The main difference is that characters that have special meaning with a regular expression (like parenthesis, square brackets, and asterisk) must be escape with a backslash or surrounded with double quotes. Also, a period (\texttt{.}) can be used to match any character at all, which is helpful for catching error conditions.

Figure 3.6 shows a simple but complete example to get you started. This specification describes just a few tokens: a single character addition (which must be escaped with a backslash), the \texttt{while} keyword, an identifier consisting of one or more letters, and a number consisting of one or more digits. As is typical in a scanner, any other type of character is an error, and returns an explicit token type for that purpose.

Flex generates the scanner code, but not a complete program, so you must write a \texttt{main} function to go with it. Figure 3.7 shows a simple driver program that uses this scanner. First, the main program must declare as \texttt{extern} the symbols it expects to use in the generated scanner code: \texttt{yyin} is the file from which text will be read, \texttt{yylex} is the function that implements the scanner, and the array \texttt{yytext} contains the actual text of each token discovered.

Finally, we must have a consistent definition of the token types across the parts of the program, so into \texttt{token.h} we put an enumeration describing the new type \texttt{token_t}. This file is included in both \texttt{scanner.flex} and \texttt{main.c}.

Figure 3.9 shows how all the pieces come together. \texttt{scanner.flex} is
### Contents of File: scanner.flex

```flex
%
#include "token.h"
%
DIGIT [0-9]
LETTER [a-zA-Z]
%
(" 	
) /* skip whitespace */
\+    { return TOKEN_ADD; }
while    { return TOKEN_WHILE; }
{LETTER}+    { return TOKEN_IDENT; }
{DIGIT}+    { return TOKEN_NUMBER; }
.    { return TOKEN_ERROR; }
%
int yywrap() { return 1; }
```

Figure 3.6: Example Flex Specification

### Contents of File: main.c

```c
#include "token.h"
#include <stdio.h>

extern FILE *yyin;
extern int yylex();
extern char *yytext;

int main() {
    yyin = fopen("program.c","r");
    if(!yyin) {
        printf("could not open program.c\n");
        return 1;
    }

    while(1) {
        token_t t = yylex();
        if(t==TOKEN_EOF) break;
        printf("token: %d text: %s\n",t,yytext);
    }
}
```

Figure 3.7: Example Main Program

*DRAFT September 12, 2016*
3.7. Practical Considerations

Handling keywords. - In many languages, keywords (such as `while` or `if`) would otherwise match the definitions of identifiers, unless specially handled. There are several solutions to this problem. One is to enter a regular expression for every single keyword into the Flex specification. (These must precede the definition of identifiers, since Flex will accept the first expression that matches.) Another is to maintain a single regular expression that matches all identifiers and keywords. The action associated with that rule can compare the token text with a separate list of keywords and return the appropriate type. Yet another approach is to treat all keywords and identifiers as a single token type, and allow the problem to be sorted out by the parser. (This is necessary in languages like PL/1, where
identifiers can have the same names as keywords, and are distinguished by context.)

**Tracking source locations.** In later stages of the compiler, it is useful for the parser or typechecker to know exactly what line and column number a token was located at, usually to print out a helpful error message. (“Undefined symbol spider at line 153.”) This is easily done by having the scanner match newline characters, and increase the line count (but not return a token) each time one is found.

**Cleaning tokens.** Strings, characters, and similar token types need to be cleaned up after they are matched. For example, "hello\n" needs to have its quotes removed and the backslash-n sequence converted to a literal newline character. Internally, the compiler only cares about the actual contents of the string. Typically, this is accomplished by writing a function `string_clean` in the postamble of the Flex specification. The function is invoked by the matching rule before returning the desired token type.

**Constraining tokens.** Although regular expressions can match tokens of arbitrary length, it does not follow that a compiler must be prepared accept them. There would be little point to accepting a 1000-letter identifier, or an integer larger than the machine’s word size. The typical approach is to set the maximum token length (`YYLMAX` in flex) to a very large value, then examine the token to see if it exceeds a logical limit in the action that matches the token. This allows you to emit an error message that describes the offending token as needed.

**Error Handling.** The easiest approach to handling errors or invalid input is simply to print a message and exit the program. However, this is unhelpful to users of your compiler – if there are multiple errors, it’s (usually) better to see them all at once. A good approach is to match the minimum amount of invalid text (using the dot rule) and return an explicit token type indicating an error. The code that invokes the scanner can then emit a suitable message, and then ask for the next token.