2. Scanning Tokens

2.1 Introduction

The first step in compiling a program is to break it into tokens. For example, given the *j--* program

```
package pass;

import java.lang.System;

public class Factorial
{
    // Two methods and a field

    public static int factorial(int n)
    {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args)
    {
        int x = n;
        System.out.println(x + "! = " + factorial(x));
    }

    static int n = 5;
}
```

we want to produce the sequence of tokens

```
package, pass, ;, import, java, ., lang, ., System, ;, public, class,
Factorial, {, public, static, int, factorial, (, int, n,), {, if, (, n,
<=, 0,), return, 1, ;, else, return, n, *, factorial, (, n, -, 1,), ;,
}, public, static, void, main, (, String, [, ], args,), {, int, x, =,
n, ;, System, ., out, ., println, (, x, +, "! =" , +, factorial, (,
x,),), ;, }, static, int, n, =, 5, ;,
```

and `}`.

Notice how we have broken down the program's text into its component elements. We call these, the *lexical tokens* (or lexical elements) that are described by the language's *lexical syntax*. Notice also how the comment has been ignored in the sequence of tokens; this is because comments have no meaning to the compiler or to the results of executing the program.

The lexical syntax of a programming language is usually described separately. For example, in *j--* we describe identifiers as "a letter, underscore, or dollar-sign, followed by

zero or more letters, digits, underscores or dollar-signs". We can also describe these tokens more formally, using what are called *regular expressions.* We visit regular expressions later in section 2.3. For now, let's be a little less formal.

In describing our lexical tokens we usually separate them into categories. In our example program, **`public`**, **`class`**, **`static`**, and **`void`** are categorized as *reserved words*. **`Factorial, main, String, args, System, out,`** and **`println`** are all *identifiers*; an identifier is a token in its own right. The string **`" = "`** is a *literal*, a *string literal* in this instance. The rest are *operators* and *separators*; notice that we distinguish between single-character operators such as "**`+`**", and multi-character operators like "**`>=`**". In describing the tokens, we can usually simply list the different reserved words and operators, and describe the structure of identifiers and literals.

2.2 Scanning Tokens

We call the program that breaks the input stream of characters into tokens, a *lexical analyzer* or, less formally a *scanner*.

A scanner may be handcrafted, that is a program written by the compiler writer; or it may be generated automatically from a specification consisting of a sequence of regular expressions. The lexical analyzer that we describe in this section will be handcrafted. We look at generated scanners later.

When we look at the problem of producing a sequence of tokens from a program like that above, we must determine where each token begins and ends. Clearly, *white space* (blanks, tabs, new lines, etc.) plays a role; in the example above it is used to separate the reserved word **`public`** from **`class`**, and **`class`** from the identifier **`Factorial`**. But not all tokens are necessarily separated by white space. Whether or not we have come to the end of a token in scanning an input stream of characters depends on what sort of token we are currently scanning. For example, in the context of scanning an identifier, if we come across a letter, that letter is clearly part of the identifier. On the other hand, if we are in the process of scanning an integer value (consisting of the digits 0 to 9), then that same letter would indicate that we have come to the end of our integer token. For this reason, we find it useful to describe our scanner using a *state transition diagram*.

Identifiers and integer literals

For example, consider the state transition diagram in Figure 2.1. It may be used for recognizing *j--* identifiers and decimal integers.

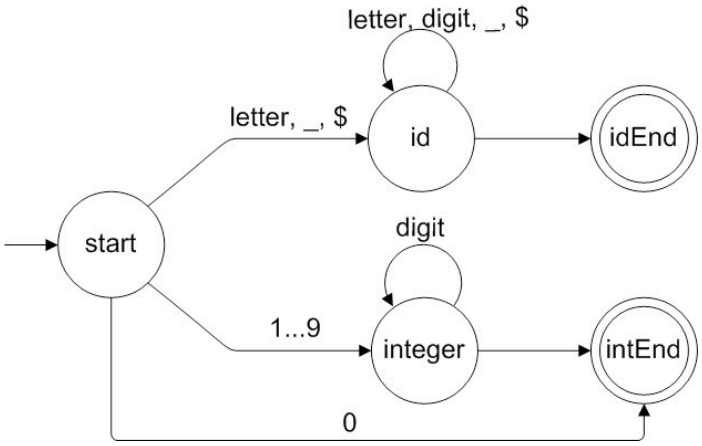


Figure 2.1 State Transition Diagram for Identifiers and Integers

In a state transition diagram, the nodes represent states, and directed edges represent moves from one state to another depending on what character has been scanned. If a character scanned does not label any of the edges, then the unlabeled edge is taken (and the input is not advanced). Think of it as a machine that recognizes (scans) identifiers and integer literals.

Consider the case when the next token in the input is the identifier **`x1`**. Beginning in the *start* state, the machine considers the first character, **`x`**; because it is a letter, the machine scans it and goes into the *id* state (that is, a state in which we are recognizing an identifier). Seeing the next **`1`**, it scans that digit and goes back into the *id* state. When the machine comes across a character that is neither a letter, nor a digit, nor an underscore, nor a dollar sign, it takes the unmarked edge and goes into the *idEnd* state (a final state indicated by a double circle) without scanning anything.

If the first character were the digit 0 (zero), the machine would scan the zero and go directly into the (final) *intEnd* state. On the other hand, if the first character were a non-zero digit, it would scan it and go into the *integer* state. From there it would scan succeeding digits and repeatedly go back into the *integer* state until a non-digit character is reached; at this point the machine would go into the (final) *intEnd* state without scanning another character.

An advantage of basing our program on such a state transition diagram is that it takes account of *state* in deciding what to do with the next input character. Clearly, what the scanner does with an incoming letter depends on whether it is in the *start* state (it would go into the id state), the *id* state (it would remain there), or the *integer* state (where it would go into the *intEnd* state, recognizing that it has come to the end of the integer).

It is relatively simple to implement a state transition diagram in code. For example, the code for our diagram above might look something like

```
if (isLetter(ch) || ch == '_' || ch == '$'){
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) ||
           ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(),
                         line);
}
else if (ch == '0') {
    nextCh();
    return new TokenInfo(INT_LITERAL, "0", line);
}
else if (isDigit(ch)){
    buffer = new StringBuffer();
    while (isDigit(ch)) {
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(INT_LITERAL, buffer.toString(),
                         line);
}
```

Choices translate to if-statements and cycles translate to while-statements. Notice that the **`TokenInfo`** object encapsulates the value of an integer as the string of digits denoting it. For example, the number 6449 is represented as the String, "6449". Translating this to binary is done later, during code generation.

In the code above, **`TokenInfo`** is the type of an object that encapsulates a representation of the token found, its image (if any), and the number of the line on which it was found.

Reserved words

There are two ways for recognizing reserved words. In the first, we complicate the state transition diagram for recognizing reserved words, and distinguishing them from (non-reserved) identifiers. For example, the state transition diagram fragment in figure 2.2 recognizes reserved words (and identifiers) beginning with the letter 'i' or the letter 'n'.

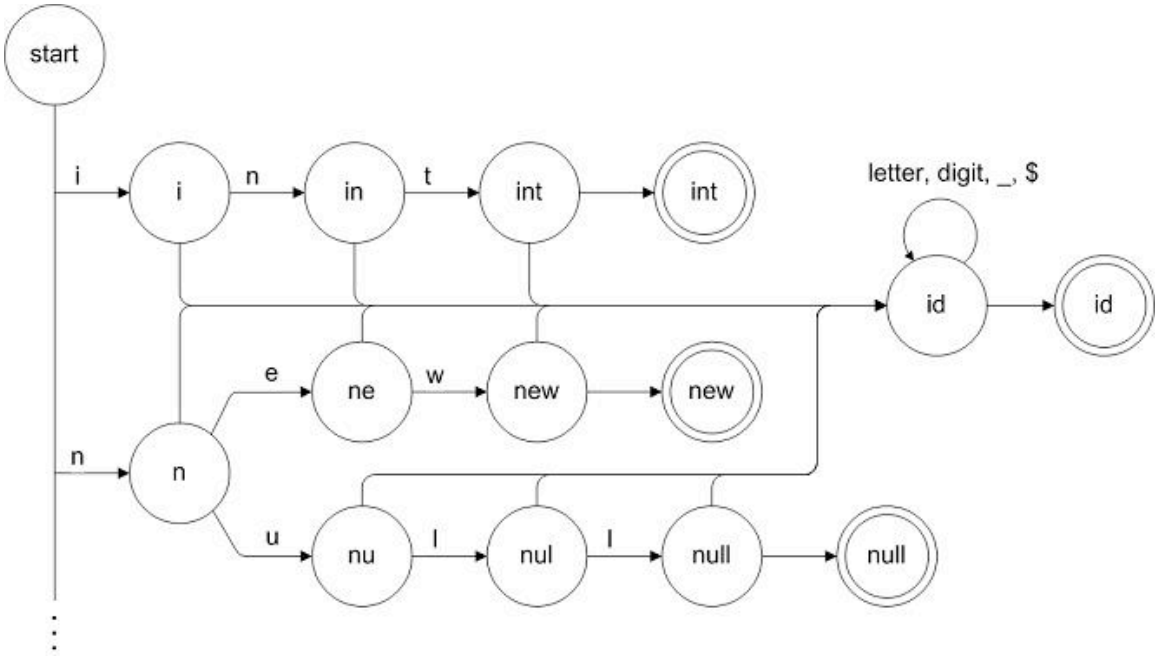


Figure 2.2 A State Transition Diagram that Distinguishes Reserved Words from Identifiers

The code corresponding to this might look something like

```
…
else if (ch == 'n') {
    buffer.append(ch);
    nextCh();
    if (ch == 'e') {
        buffer.append(ch);
        nextCh();
            if (ch == 'w') {
            buffer.append(ch);
            nextCh();
            if (!isLetter(ch) && !isDigit(ch) &&
                     ch != '_' && ch !=  '$') {
                return new TokenInfo(NEW, line);
            }
        }
    }
    else if (ch == 'u') {
        buffer.append(ch);
        nextCh();
        if (ch == 'l') {
            buffer.append(ch);
            nextCh();
            if (ch == 'l') {
                buffer.append(ch);
                nextCh();
                if (!isLetter(ch) && !isDigit(ch) &&
```

```
                        ch != '_' && ch !=  '$') {
                    return new TokenInfo(NULL, line);
                }
            }
        }
    }
    while (isLetter(ch) || isDigit(ch) ||
            ch == '_' || ch == '$') {
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(),
                         line);
}
else …
```

Unfortunately, such state transition diagrams and the corresponding code are too complex.[1] Imagine the code necessary for recognizing all reserved words.

A second way is much more straightforward for handwritten token scanners. We simply recognize a simple identifier, which may or may not be one of the reserved words. We look that identifier up in a table of reserved words. If it is there in the table, then we return the corresponding reserved word. Otherwise, we return the (non-reserved) identifier. The state transition diagram then looks something like that in Figure 2.3.

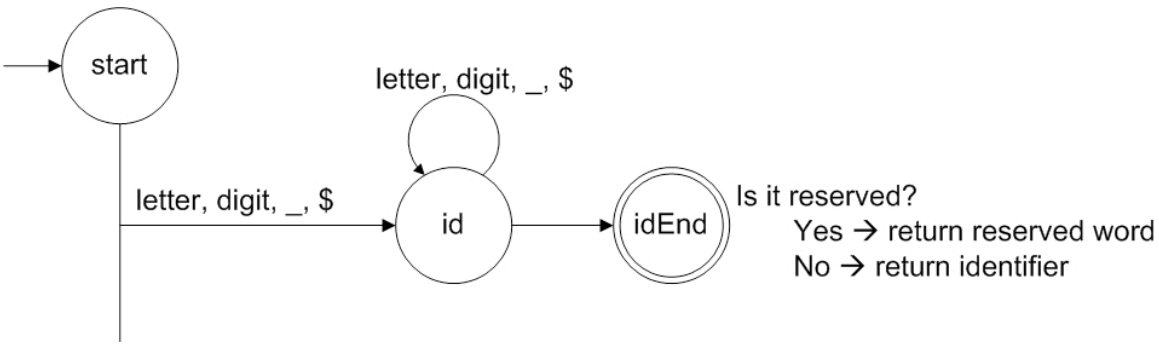


Figure 2.3 Recognizing Words and Looking Them up in a Table to See If They are Reserved

The code corresponding to this logic would look something like the following.

```
    if (isLetter(ch) || ch == '_' || ch == '$') {
```

[1] Unless the diagrams (and corresponding code) are automatically generated from a specification; we do this later in the chapter when we visit regular expressions and finite state automata.

```
        buffer = new StringBuffer();
        while (isLetter(ch) || isDigit(ch) ||
               ch == '_' || ch == '$'){
            buffer.append(ch);
            nextCh();
        }
        String identifier = buffer.toString();
        if (reserved.containsKey(identifier)) {
            return new TokenInfo(reserved.get(identifier),
                                 line);
        }
        else {
            return new TokenInfo(IDENTIFIER, identifier,
                                 line);
        }
    }
```

This relies on a map, **reserved**, mapping reserved identifiers to their representations:

```
reserved = new Hashtable<String, Integer>();
reserved.put("abstract", ABSTRACT);
reserved.put("boolean", BOOLEAN);
reserved.put("char", CHAR);
…
reserved.put("while", WHILE);
```

We follow this latter method, of looking up identifiers in a table of reserved words, in our handwritten lexical analyzer.

Operators

The state transition diagram deals nicely with operators. We must be careful to watch for certain multi-character operators. For example, the state transition diagram fragment for recognizing tokens beginning with ';', '=', '!', or '*' would look like that in Figure 2.4.

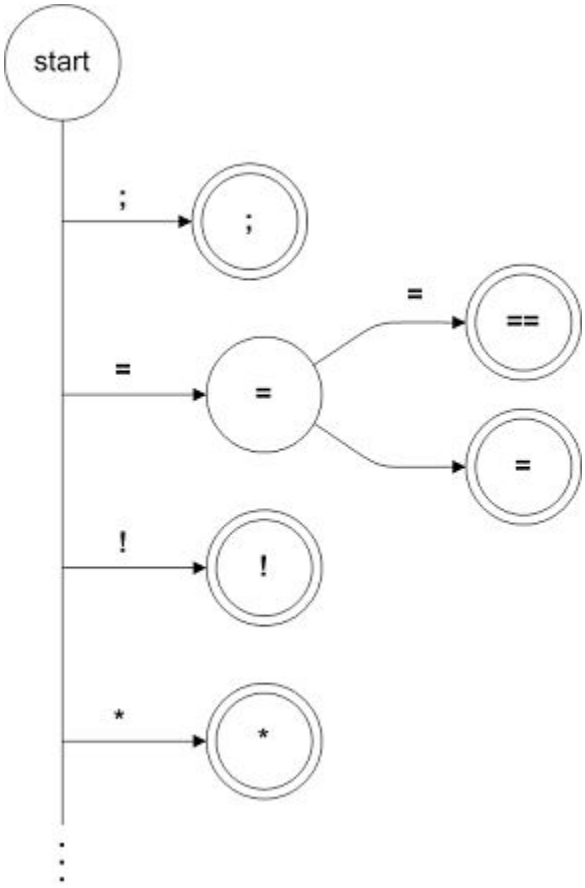


Figure 2.4 A State Transition Diagram for Recognizing the Separator ";" and the Operators "==", "=", "!", and "*"

The code corresponding to the state transition diagram in Figure 2.4 would look like the following. Notice the use of the switch-statement for deciding among first characters.

```
switch (ch) {
…
case ';':
    nextCh();
    return new TokenInfo(SEMI, line);
case '=':
    nextCh();
    if (ch == '=') {
        nextCh();
        return new TokenInfo(EQUAL, line);
    }
    else {
        return new TokenInfo(ASSIGN, line);
    }
case '!':
    nextCh();
```

```
    return new TokenInfo(LNOT, line);
case '*':
    nextCh();
    return new TokenInfo(STAR, line);
…
}
```

White space

Before attempting to recognize the next incoming token, one wants to skip over all white space. In *j--*, as in Java, white space is defined as the ASCII SP characters (spaces), HT (horizontal tabs), FF (form feeds) and line terminators; in *j--* (as in Java) we can denote these characters as ' ', '\t', 'f', '\b', '\r' and '\n' respectively. Skipping over white space is done from the start state, as illustrated in Figure 2.5.

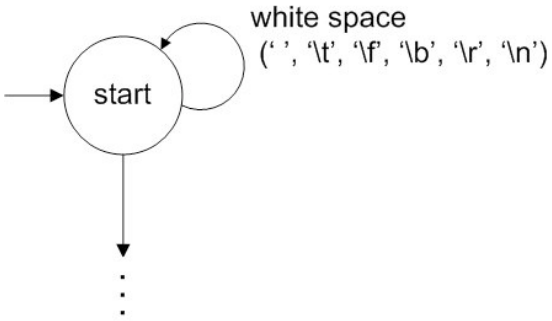


Figure 2.5 Dealing with White Space

The code for this is simple enough, and comes at the start of a method for reading the next incoming token:

```
while (isWhitespace(ch)) {
    nextCh();
}
```

Comments

Comments can be considered a special form of white space because the compiler ignores them. A *j--* comment extends from a double-slash, // to the end of the line. This complicates the skipping of white space somewhat, as illustrated in Figure 2.6.

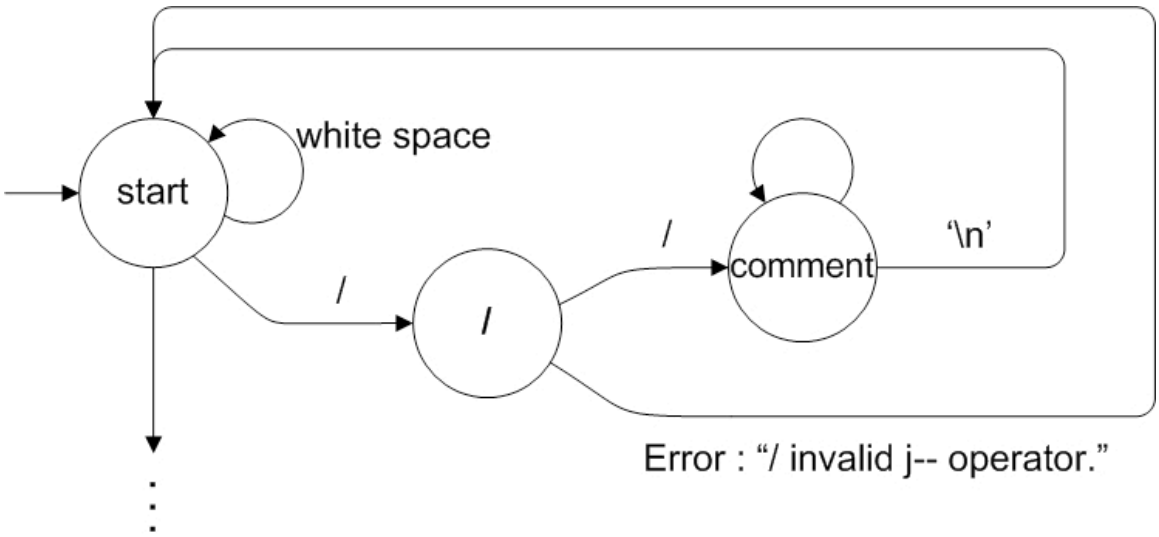


Figure 2.6 Treating one-line (//…) Comments as White Space

Notice that a / operator on its own is meaningless in *j*--. Adding it (for denoting division) is left as an exercise. But notice that when coming upon an erroneous single /, the lexical analyzer reports the error and goes back into the start state in order to fetch the next valid token. This is all captured in the code:

```
boolean moreWhiteSpace = true;
while (moreWhiteSpace) {
    while (isWhitespace(ch)) {
        nextCh();
    }
    if (ch == '/') {
        nextCh();
        if (ch == '/') {
            // CharReader maps all new lines to '\n'
            while (ch != '\n' && ch != EOFCH) {
                nextCh();
            }
        }
        else {
            reportScannerError("Operator / is not supported in j--.");
        }
    }
    else {
        moreWhiteSpace = false;
    }
}
```

There are other kinds of tokens we must recognize as well, for example String literals and character literals. The code for recognizing all tokens appears in file **`Scanner.java`**; the principal method of interest is **`getNextToken()`**. This file is part of source code of the *j*-- compiler that we discussed in Chapter 1. At the end of this chapter you will find

exercises that ask you to modify this code (as well as that of other files) for adding tokens and other functionality to our lexical analyzer.

A pertinent quality of the lexical analyzer described here is that it is handcrafted. Although writing a lexical analyzer by hand is relatively easy, particularly if it is based on a state transition diagram, it is prone to error. In the next section we shall learn how we may automatically produce a lexical analyzer from a notation based on regular expressions.

2.3 Regular Expressions

Regular expressions comprise a relatively simple notation for describing patterns of characters in text. For this reason, one finds them in text processing tools such as text editors. We are interested in them here because they are also convenient for describing lexical tokens.

Definition 2.1. We say that a *regular expression* defines a *language* of strings over an *alphabet*. Regular expressions may take one of the following forms:

1. If *a* is in our alphabet, then the regular expression *a* describes the language consisting of the string "a". We call this language L(*a*).

2. If *r* and *s* are regular expressions then their *concatenation rs* is also a regular expression describing the language of all possible strings obtained by concatenating a string in the language described by *r*, to a string in the language described by *s*. We call this language L(*rs*).

3. If *r* and *s* are regular expressions then the *alternation r*|*s* is also a regular expression describing the language consisting of all strings described by either *r* or *s*. We call this language L(*r*|*s*).

4. If *r* is a regular expression, the *repetition*[2] *r** is also a regular expression describing the language consisting of strings obtained by concatenating zero or more instances of strings described by *r* together. We call this language L(*r**).

   Notice that,
   $r^0 = \varepsilon$, the empty string of length 0.
   $r^1 = r$
   $r^2 = rr$
   $r^3 = rrr$, and so on …

   *r** denotes an infinite number of finite strings.

[2] Also known as the *Kleene closure*.

5. ε is a regular expression describing the language containing only the empty string.

6. Finally, if *r* is a regular expression, then (*r*) is also a regular expression denoting the same language. The parentheses serve only for grouping.

Examples

So, for example, given an alphabet {0,1},

	0	is a regular expression describing the single string 0
	1	is a regular expression describing the single string 1
	0|1	is a regular expression describing the language of two strings 0 and 1
	(0|1)	is a regular expression describing the (same) language of two strings 0 and 1
(2.1)	(0|1)*	is a regular expression describing the language of all strings, including the empty string, of 1's and 0's: ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, …, 000111,…
	1(0|1)*	is a regular expression describing the language of all strings of 1's and 0's that start with a 1.
	0|1(0|1)*	is a regular expression describing the language consisting of all binary numbers (excluding those having unnecessary leading zeros).

Notice that there is an order of precedence in the construction of regular expressions: repetition has the highest precedence, then concatenation, and finally alternation. So, 01*0|1* is equivalent to (0(1*)0)|(1*). Of course, parentheses may always be used to change the grouping of sub-expressions.

Examples

Given an alphabet {a,b},

(2.2) a(a|b)*

denotes the language of non-empty strings of a's and b's, beginning with an a. The regular expression,

(2.3) aa | ab | ba | bb

denotes the language of all two-symbol strings over the alphabet, and,

(2.4) (a|b)*ab

denotes the language of all strings of a's and b's, ending in ab (this include the string ab itself).

As in programming, we often find it useful to give names to things.

D = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Then we can say, 0 | D(D|0)* denotes the language of natural numbers. Of course, this is just the same as

0 | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0)*

There are all sorts of extensions to the notation of regular expressions, all of which are shorthand for standard regular expressions. For example,

[0..9] is shorthand for (0|1|2|3|4|5|6|7|8|9), and

(2.5)

[a..z] is shorthand for (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)

In describing the lexical tokens of programming languages, one uses some standard input character set as one's alphabet, for example, the 128 character ASCII set, the 256 character extended ASCII set, or the much larger Unicode set. Java works with Unicode, but aside from identifiers, characters and String literals, all input characters are ASCII, making implementations compatible with legacy operating systems. We do the same for *j--*.

Examples

The reserved words may be described simply by listing them. For example,

```
  abstract
| boolean
| char
.
.
.
| while
```

(2.6)

Likewise for operators. For example,

```
  =
| ==
| >
.
.
.
| *
```

(2.7)

Identifiers are easily described; for example,

(2.8) ([a..zA..Z] | _ | $)([a..zA..Z0..9] | _ | $)*

which is to say, an identifier begins with a letter, an underscore, or a dollar sign, followed by zero or more letters, digits, underscores and dollar signs.

A full description of the lexical syntax may be found in Appendix B. In the next section, we formalize state transition diagrams.

2.4 Finite State Automata

It turns out that for any language described by a regular expression, there is a state transition diagram that can parse strings in this language. These are called finite state automata.

Definition 2.2. A *finite state automaton* (FSA) F is a quintuple

FSA F = (Σ, S, s_0, M, F) where

- Σ (pronounced sigma) is the input alphabet.
- S is a set of states.
- $s_0 \in$ S is a special start state.
- M is a set of moves or state transitions of the form

  $$M(r, a) = s \text{ where } r,s \in S, a \in \Sigma$$

  Read as, "if one is in state r, and the next input symbol is a, scan the a and move into state s.
- F $\subset$ S is a set of final states.

A finite state automaton is just a formalization of the state transition diagrams we saw in section 2.2. We say that a finite state automaton *recognizes* a language. A sentence over the alphabet Σ is said to be in the language recognized by the FSA if, starting in the start state, a set of moves based on the input takes us into one of the final states.

Example

Consider the regular expression,

(2.9) (a|b)a*b

This describes a language over the alphabet {a, b}; it is the language consisting of all strings starting with either an a or a b, followed by zero or more a's, and ending with a b. An FSA that recognizes this same language is

FSA $F = (\Sigma, S, s_0, M, F)$ where

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

$S = \{0, 1, 2\}$

$s_0 = 0$

(2.10) $M = \{ M(0, a) = 1,$
$M(0, b) = 1,$
$M(1, a) = 1,$
$M(1, b) = 2 \}$

$F = \{ 2 \}$

The corresponding state transition diagram is shown in Figure 2.7.

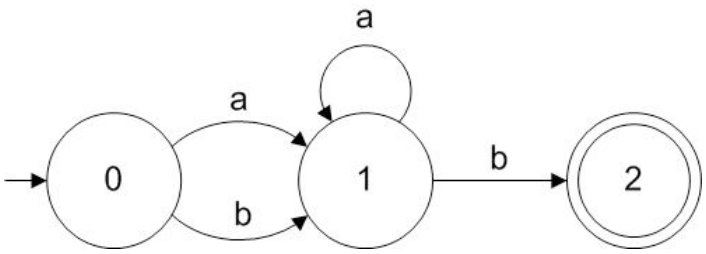


Figure 2.7 A FSA recognizing (a|b)a*b

A FSA recognizes strings in the same way that state transition diagrams do. For example, given the input sentence

baaab

and beginning in the start state 0, the following moves are prescribed:

M(0, b) = 1 in state 0 we scan a b and go into state 1,
M(1, a) = 1 in state 1 we scan an a and go back into state 1,
M(1, a) = 1 in state 1 we scan an a and go back into state 1 (again),
M(1, a) = 1 in state 1 we scan an a and go back into state 1 (again), and
M(1, b) = 2 finally, in state 1 we scan a b and go into the final state 2.

Each move scans the corresponding character. Because we end up in a final state after scanning the entire input string of characters, the string is accepted by our FSA.

The question arises, given the regular expression in (2.9), have we a way of automatically generating the FSA in (2.10)? The answer is yes! But first we must discuss two categories of automata: Non-deterministic Finite-state Automata (NFA) and Deterministic Finite-state Automata (DFA).

2.5 Non-deterministic Finite-state Automata (NFA) vs. Deterministic Finite-state Automata (DFA)

The example FSA given above (2.10) is actually a *deterministic* finite-state automaton.

Definition 2.3. A *deterministic finite-state automaton* (DFA) is an automaton where there are no ε-moves (see below), and there is a unique move from any state, given a single input symbol a. That is, there **cannot** be two moves:

$$M(r, a) = s$$
$$M(r, a) = t$$

where $s \neq t$. So, from any state there is at most one state that we can go into, given an incoming symbol.

Definition 2.4. A *non-deterministic finite-state automaton* (NFA) is a finite state automaton that allows either of the following conditions.

- More than one move from the same state, on the same input symbol, i.e.,
  $M(r, a) = s$,
  $M(r, a) = t$, for states r, s and t where $s \neq t$.
- An ε-move defined on the empty string ε, i.e.,
  $M(r, \varepsilon) = s$,
  which says we can move from state r to state s without scanning any input symbols.

An example of a non-deterministic finite-state automaton is

NFA $N = (\Sigma, S, s_0, M, F)$ where

$\Sigma = \{a, b\}$
$S = \{0, 1, 2\}$
$s_0 = 0$

(2.11) $M = \{ M(0, a) = 1,$
$M(0, b) = 1,$
$M(1, b) = 1,$
$M(1, \varepsilon) = 0,$
$M(1, b) = 2 \}$

$F = \{ 2 \}$

and is illustrated by the diagram in Figure 2.8. This NFA recognizes all strings of a's and b's that begin with an a and end with a b. Like any FSA, an NFA is said to recognize an input string if, starting in the start state, there exist a set of moves based on the input that takes us into one of the final states.

But this automaton (2.11) is definitely not deterministic. Being in state 1 and seeing a b, we can go either back into state 1 or into state 2. Moreover the automaton has an ε-move.

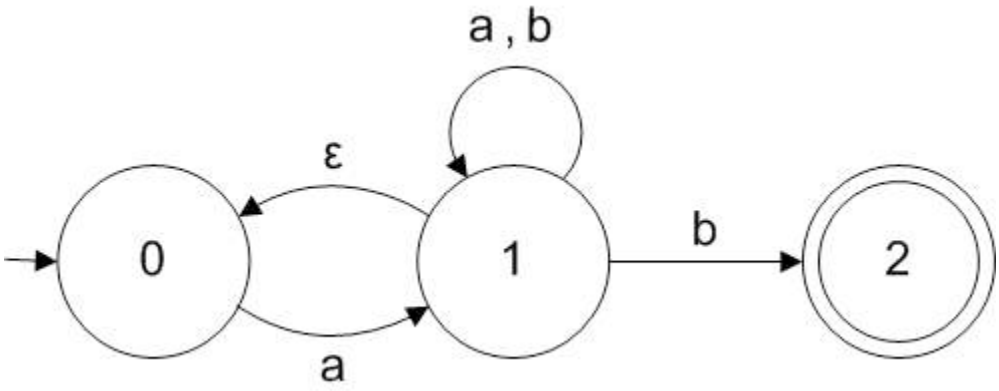


Figure 2.8 An NFA

Needless to say, a lexical analyzer based on a non-deterministic finite state automaton requires backtracking, where one based on a deterministic finite-state automaton does not. One might ask why we are at all interested in NFA. Our only interest in non-deterministic finite-state automata is that they are an intermediate step from regular expressions to deterministic finite-state automata.

2.6 Regular Expressions to NFA

Given any regular expression R, we can construct a non-deterministic finite state automaton N that recognizes the same language; that is, L(N) = L(R). We show that this is true by using what is called *Thompson's construction*:

1. If the regular expression r takes the form of an input symbol, a, then the NFA that recognizes it has two states: a start state and a final state, and a move on symbol a from the start state to the final state.

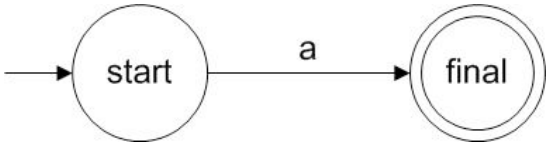


Figure 2.9 Parsing Symbol 'a'

2. If N_r and N_s are NFA recognizing the languages described by the regular expressions r and s respectively, then we can create a new NFA recognizing the language described by rs as follows. We define an ε-move from the final state of N_r to the start state of N_s. We then choose the start state of N_r to be our new start state, and the final state of N_s to be our new final state.

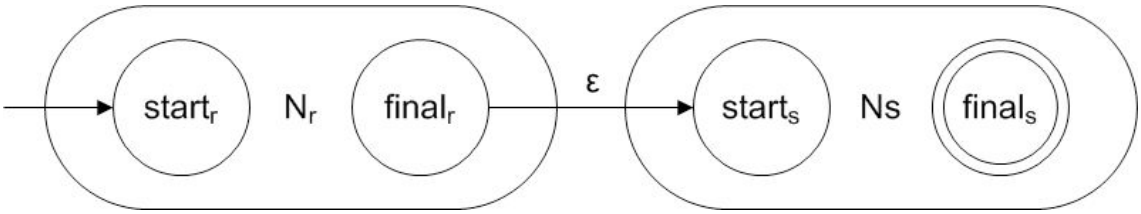


Figure 2.10 Concatenation

3. If N_r and N_s are NFA recognizing the languages described by the regular expressions r and s respectively, then we can create a new NFA recognizing the language described by r|s as follows. We define a new start state, having ε-moves to each of the start states of N_r and N_s, and we define a new final state and add ε-moves from each of N_r and N_s to this state.

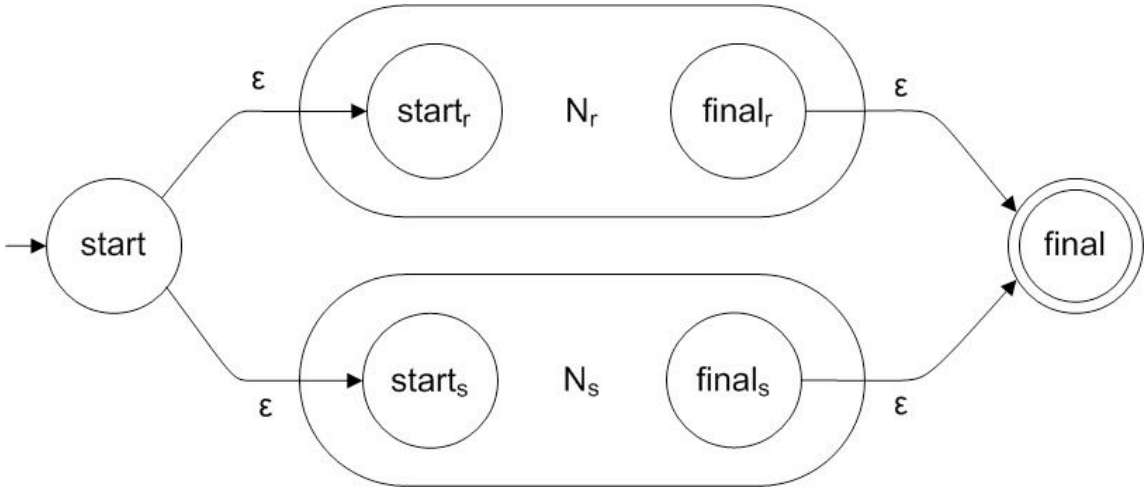


Figure 2.11 Alternation

4. If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing r* as follows. We add an ε-move from N_r's final state back to its start state. We define a new start state and a new final state, and we add ε-moves from the new start state to both N_r's start state and the new final state, and we define an ε-move from N_r's final state to the new final state.

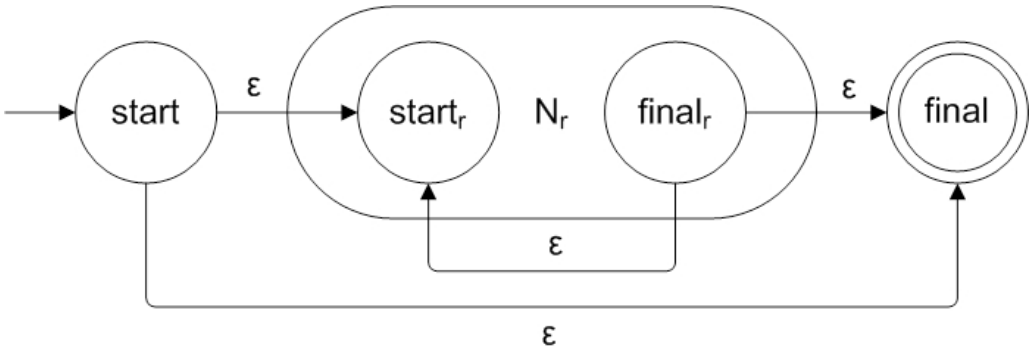


Figure 2.12 Repetition

5. If r is ε then we just need an ε-move from the start state to the final state.

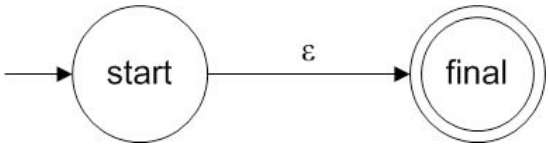


Figure 2.13 ε-move

6. If N_r is our NFA recognizing the language described by r, then N_r also recognizes the language described by (r). (Parentheses only group expressions.)

Example

As an example, reconsider the regular expression from (2.9):

(2.12) (a|b)a*b

We decompose this regular expression, and display its syntactic structure in Figure 2.14.

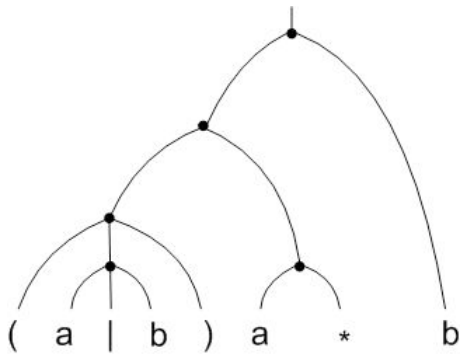
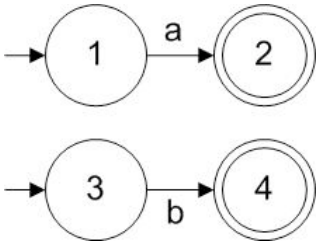


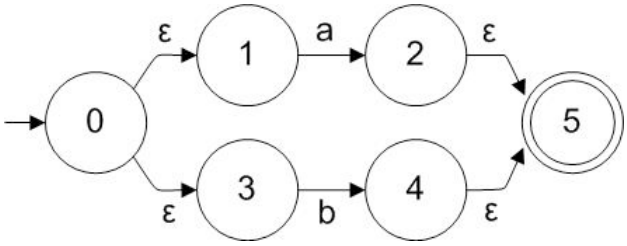
Figure 2.14 The syntactic structure for (a|b)a*b

We can construct our NFA based on this structure, beginning with the simplest components, and putting them together according to the six rules above.

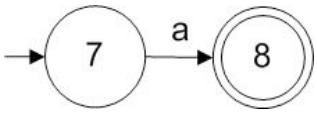
1. We start with the first a and b; the automata recognizing these are easy enough to construct using rule 1 above.



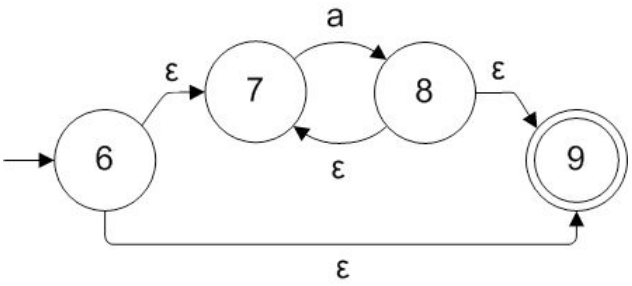
2. We then put them together using rule 3 to produce an NFA recognizing a|b.



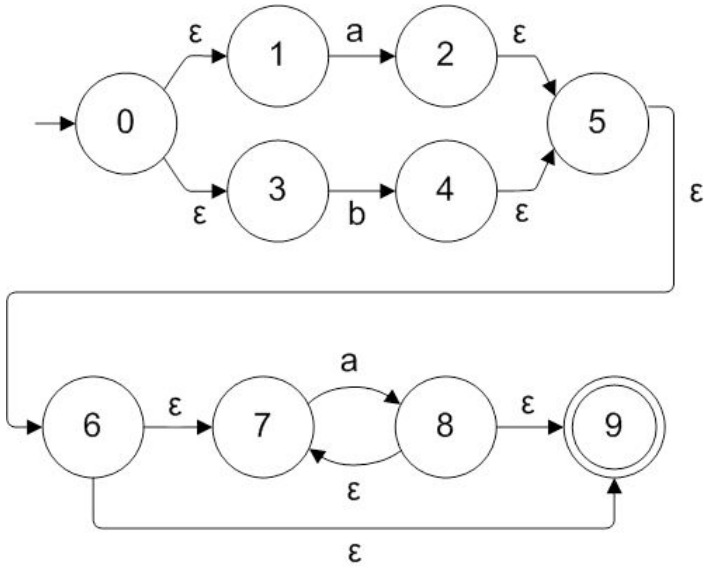
3. The NFA recognizing (a|b) is the same as that recognizing a|b, by rule 6. An NFA recognizing the second instance of a is simple enough, by rule 1 again.



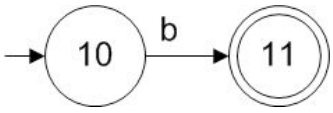
4. The NFA recognizing a* can be constructed from that recognizing a, by applying rule 4.



5. We then apply rule 2 to construct an NFA recognizing the concatenation (a|b)a*.



6. An NFA recognizing the second instance of b is simple enough, by rule 1 again.



7. Finally, we can apply rule 2 again to produce an NFA recognizing the concatenation of (a|b)a* and b, that is (a|b)a*b. This NFA is illustrated below, as Figure 2.15.

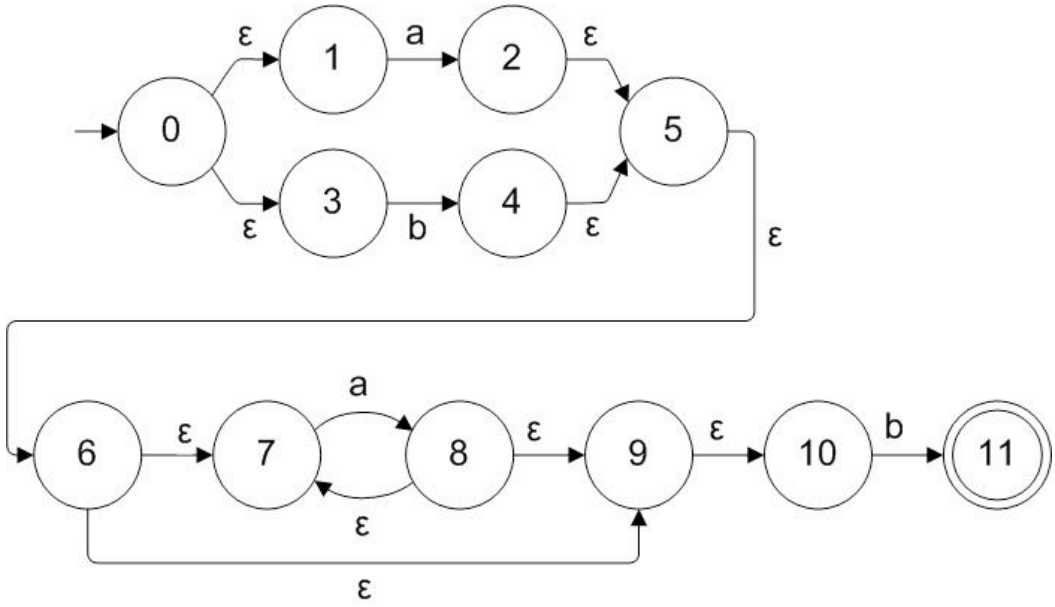


Figure 2.15 An NFA recognizing (a|b)a*b

2.7 NFA to DFA

Of course any NFA will require backtracking. This requires more time and, because we in practice wish to collect information as we recognize a token, is impractical. Fortunately, for any nondeterministic finite automaton (NFA), there is an equivalent deterministic finite automaton (DFA). By equivalent, we mean a DFA that recognizes the same language. Moreover, we can show how to construct such a DFA.

In general, the DFA that we will construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input. For this reason, we call this a *powerset construction*[3].

For example, consider the NFA constructed for (a|b)a*b, illustrated in Figures 2.15 and 2.17. The start state of our DFA, call it s_0, must reflect all the possible states that our NFA can be in before any character is scanned; that is, the NFA's start state 0, and all other states reachable from state 0 on ε-moves alone: 1 and 3. Thus, the start state in our new DFA is

[3] The technique is also known as a *subset construction*; the states in the DFA are a subset of the powerset of the set of states in the NFA

$s_0 = \{0, 1, 3\}$

This computation of all states reachable from a given state s based on ε-moves alone is called taking the ε-closure of that state.

Definition 2.5. The ε-closure(s) for a state s includes s and all states reachable from s using ε-moves alone. That is, for a state $s \in S$, ε-closure(s) = $\{s\} \cup \{ r \in S \mid$ there is a path of only ε-moves from s to r$\}$.

We will also be interested in the ε-closure over a set of states.

Definition 2.6. The ε-closure(S) for a set of states S includes s and all states reachable from any state s in S using ε-moves alone.

Algorithm 2.1 computes ε-closure(S) where S is a set of states.

Algorithm 2.1. ε-closure(S) for a set of states S

Input: a set of states, S.
Output: ε-closure(S)

```
Stack P.addAll(S);       // a stack containing all states in S
Set C.addAll(S);         // the closure initially contains the states in S
while (!P .empty() ) {
    s = P.pop();
    for (r ∈ move(s, ε)){         // move (s, ε) is a set of states
        if (r ∉ C) {
            P.push( r );
            C.add( r );
        }
    }
}
return C;
```

Given Algorithm 2.1, the algorithm for finding the ε-closure for a single state is simple. Algorithm 2.2 does this.

Algorithm 2.2. ε-closure(s) for a state s

Input: a state, s.
Output: ε-closure(s)

Set S.add(s); // S = {s}
return ε-closure(S);

Returning to our example, from the start state s_0, and scanning the symbol a, we shall want to go into a state that reflects all the states we could be in after scanning an a in the NFA: 2, and then (via ε-moves) 5, 6, 7, 9 and 10. Thus,

$M(s_0,a) = s_1$, where

s_1 = ε-closure(2) = {2, 5, 6, 7, 9, 10}

Similarly, scanning a symbol b in state s_0, we get

$M(s_0,b) = s_2$, where

s_2 = ε-closure(4) = {4, 5, 6, 7, 9, 10}

From state s_1, scanning an a, we have to consider where we could have gone from the states {2, 5, 6, 7, 9, 10} in the NFA. From state 7, scanning an a, we go into state 8, and then (by ε-moves) 7, 9, and 10. Thus,

$M(s_1,a) = s_3$, where

s_3 = ε-closure(8) = {7, 8, 9, 10}

Now, from state s_1, scanning a b, we have

$M(s_1,b) = s_4$, where

s_4 = ε-closure(11) = {11}

since there are no ε-moves out of state 11.

From state s_2, scanning an a takes us into a state reflecting 8, and then (by ε-moves) 7, 9 and 10, generating a candidate state,

{7, 8, 9, 10}

But this is a state we have already seen, namely s_3. Scanning a b, from state s_2 takes us into a state reflecting 11, generating the candidate state,

{11}

But this is s_4. Thus

$M(s_2,a) = s_3$
$M(s_2,b) = s_4$

From state s_3 we have a similar situation. Scanning an a takes us back into s_3. Scanning a b takes us into s_4. So,

$M(s_3,a) = s_3$
$M(s_3,b) = s_4$

There are no moves at all out of state s_4. So we have found all of our transitions and all of our states. Of course, the alphabet in our new DFA is the same as that in the original NFA.

But what are the final states? Since the states in our DFA mirror the states in our original NFA, any state reflecting (derived from a state containing) a final state in the NFA is a final state in the DFA. In our example, only s_4 is a final state because it contains (the final) state 11 from the original NFA.

Putting all of this together, a DFA derived from our NFA for (a|b)a*b is illustrated in Figure 2.16.

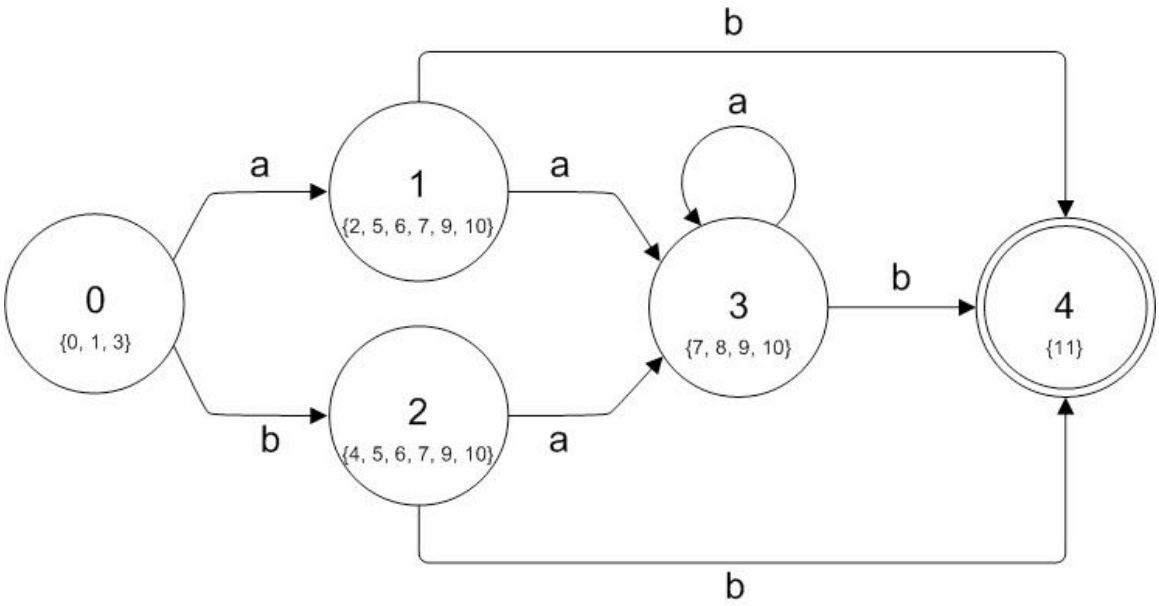


Figure 2.16 A DFA Recognizing (a|b)a*b

We can now give the algorithm for constructing a DFA that is equivalent to a NFA.

Algorithm 2.3. NFA → DFA Construction

Input: an NFA, $N = (\Sigma, S, s_0, M, F)$
Output: DFA, $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

Set S_{D0} = ε-closure(s_0);
Set S_D.add(S_{D0});
Moves M_D;
Stack stk.push(S_{D0});
i = 0;
while (!stk.empty()) {
  t = stk.pop();
  for (a : Σ) {
    S_{Di+1} = ε-closure(M(t,a));
    if (S_{Di+1} != {}) {
      if ($S_{Di+1} \notin S_D$) {
        # We have a new state.
        S_D.add(S_{Di+1});
        stk.push(S_{Di+1});
        i = i+ 1;
        M_D.add(M_D(t,a) = i_j;
      }
      else if ($\exists$ j, $S_j \in S_D \wedge S_{Di+1} == S_j$) {
        # In the case that the state already exists.
        M_D.add(M_D(t,a) = j_j;
      }
    }
  }
}
Set F_D;
for (s_D : S_D)
  for (s : s_D)
    if (s ∈ F)
      F_D.add(s_D);
return $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$;

2.8 A Minimal DFA

So, how do we come up with a smaller DFA that recognizes the same language? Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states. And, given an input string that is *not* in our language, there cannot be such a sequence; we must get stuck with no move to take or end up in a non-final state.

Clearly we must combine states if we can. Indeed, we would like to combine as many states together as we can. So the states in our new DFA are partitions of the states in the original (perhaps larger) DFA.

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA. An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states.

For example, consider the DFA from Figure 2.16, partitioned in this way. The partition into two sets of states is illustrated in Figure 2.17.

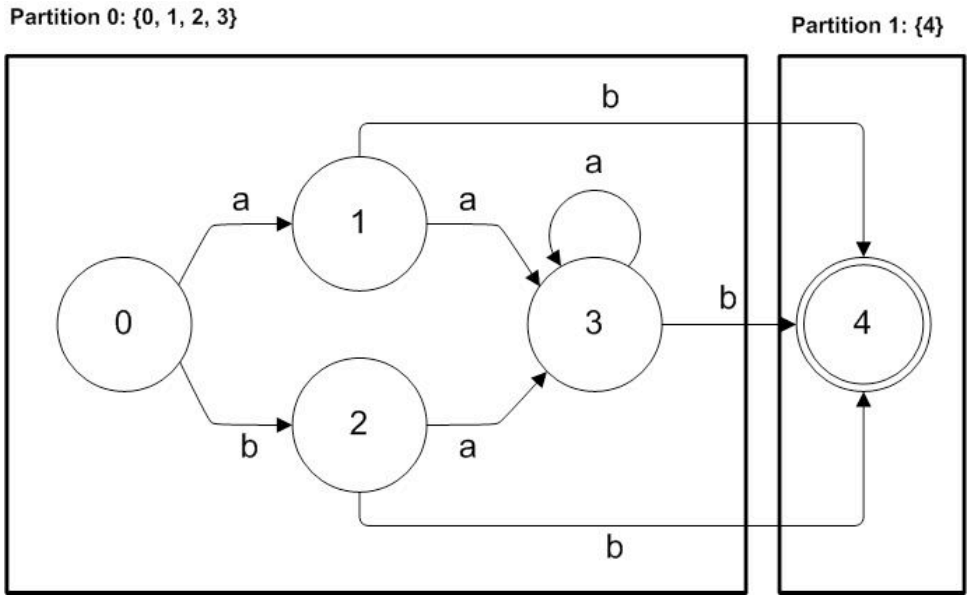


Figure 2.17 An initial partition of 2.16

The two states in this new DFA consist of the start state, {0, 1, 2, 3} and the final state {4}. Now we must make sure that, in each of these states, the move on a particular symbol reflects a move in the old DFA. That is, from a particular partition, each input symbol must move us to an identical partition.

For example, beginning in any state in the partition {0, 1, 2, 3}, an a takes us to one of the states in {0, 1, 2, 3};

Move(0,a) = 1,
Move(1,a) = 3,
Move(2,a) = 3, and
Move(3, a) = 3.

So, our partition {0, 1, 2, 3} is fine so far as moves on the symbol a are concerned. For the symbol b,

Move(0,b) = 2,

but

Move(1, b) = 4,
Move(2,b) = 4, and
Move(3, b) = 4.

So we must split the partition {0, 1, 2, 3} into two new partitions, {0} and {1, 2, 3}. The question arises, if we are in state s, and for an input symbol a in our alphabet there is no defined move,

Move(s, a) = t

What do we do? We can invent a special *dead state* d, so that we can say

Move(s, a) = d,

Thus defining moves from all states on all symbols in the alphabet.

Now, we are left with a partition into three sets: {0}, {1, 2, 3} and {4}, as is illustrated in Figure 2.18.

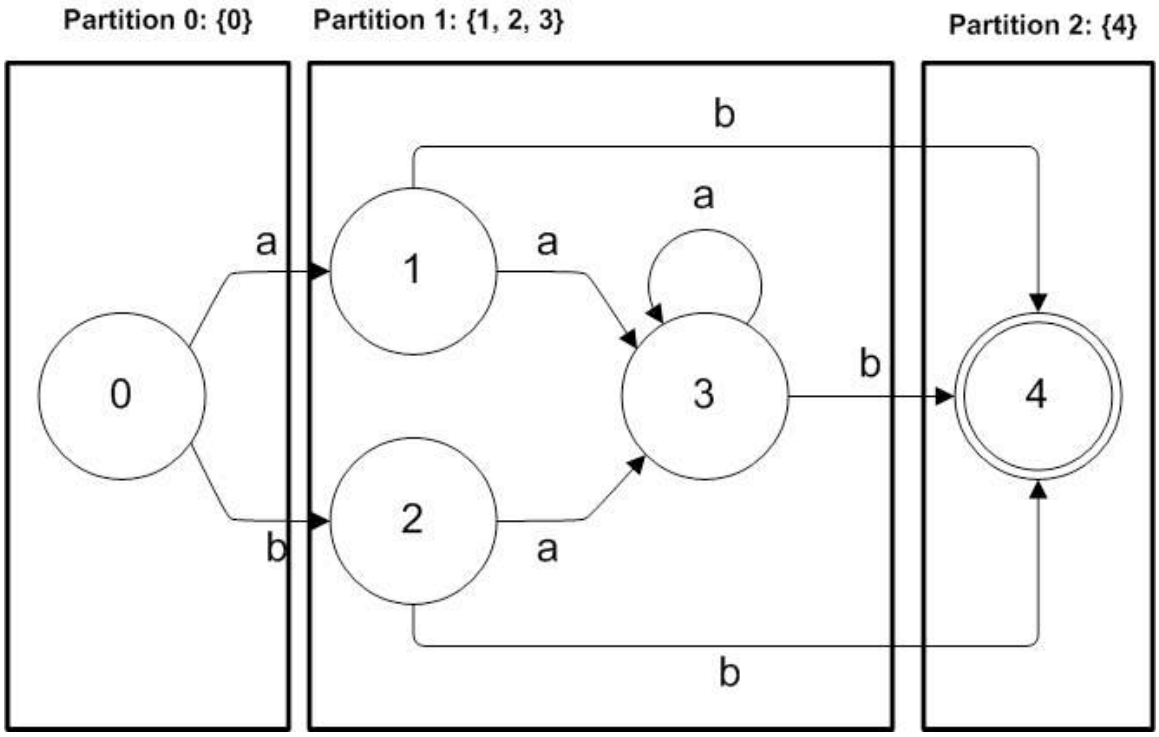


Figure 2.18 A Second Partition of 2.16

We needn't worry about {0} and {4} as they contain just one state and so correspond to (those) states in the original machine. So we consider {1, 2, 3} to see if it is necessary to split it. But, as we have seen,

Move(1,a) = 3,
Move(2,a) = 3, and
Move(3, a) = 3.

Also,

Move(1, b) = 4,
Move(2,b) = 4, and
Move(3, b) = 4.

Thus, there is no further state splitting to be done, and we are left with the smaller DFA in Figure 2.19,

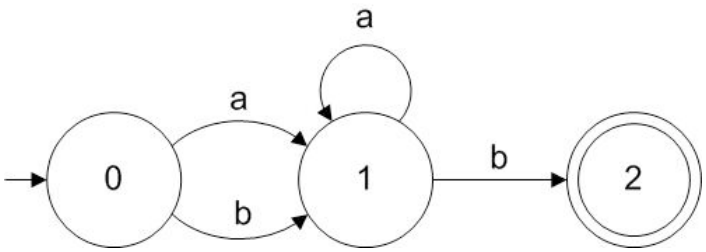


Figure 2.19 A Minimal DFA for (a|b)a*b

The algorithm for minimizing a DFA is built around this notion of splitting states.

Algorithm 2.4. Minimizing a DFA

```
Set partition = { S - F, F}  //  start with two sets: the non-final states and the final states
// Splitting the states
while (splitting occurs) {
   for (Set set : partition) {
      if (set.size() > 1) {
         for (Symbol a : Σ) {
            // Determine if  moves from this 'state' force a split
            State s = a state chosen from set
            targetSet = the set in the partition containing M(s,a)
            Set set1 = { states s from set, such that M(s,a) ∈ target set }
            Set set2 = { states s from set, such that M(s,a) ∉ target set }
            if  (set2 != {}) {
               // Yes, split the states.
               replace set in partition by set1 and set2 and break out of the for-loop
               to continue with the next set in the partition.
            }
         }
      }
   }
}
```

Then, renumber the states, and re-compute the moves for the new (possibly smaller) set of states, based on the old moves on the original set of state.

Let us quickly run through one additional example, starting from a regular expression, producing an NFA, then a DFA and finally a minimal DFA.

Example

Consider the regular expression,

(2.13) (a|b)*baa

Its syntactic structure is illustrated in Figure 2.20.

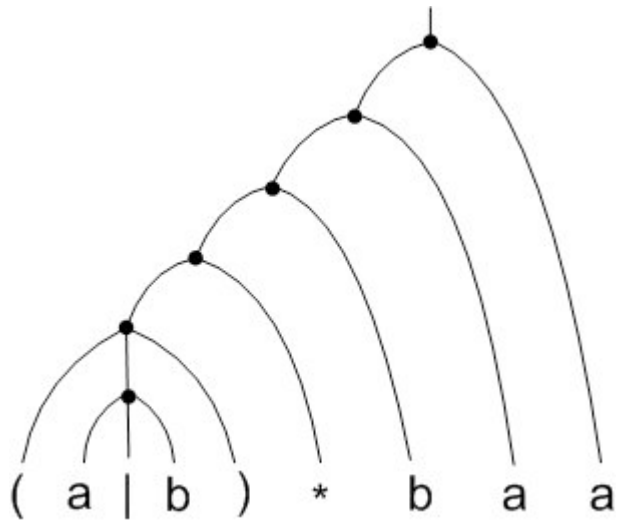


Figure 2.20 The syntactic structure for (a|b)*baa

Given this, we apply the Thompson construction for producing the NFA illustrated in Figure 2.21.

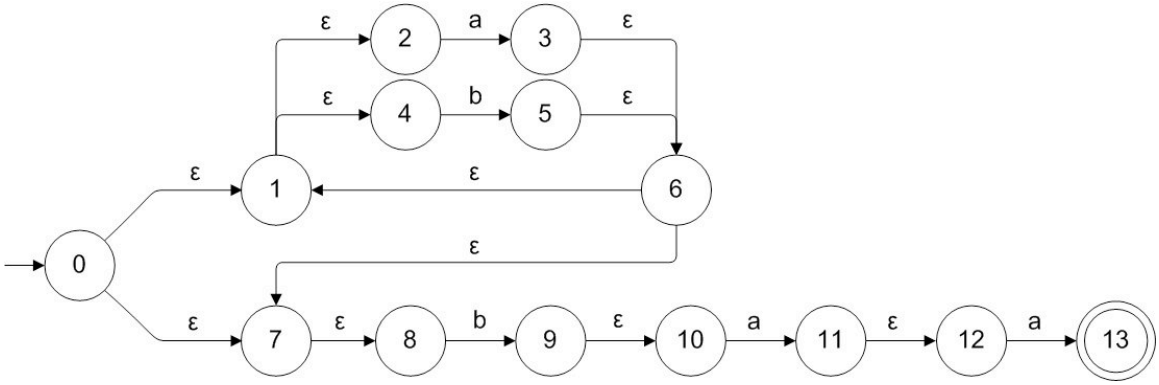


Figure 2.21 An NFA recognizing (a|b)*baa

Using the powerset construction method, we derive a DFA having the following states:

S_0: {0, 1, 2, 4, 7, 8}
M(S_0, a) : {1, 2, 3, 4, 6, 7, 8} = S_1
M(S_0, b) : {1, 2, 4, 5, 6, 7, 8, 9, 10} = S_2

$M(S_1, a)$: {1, 2, 3, 4, 6, 7, 8}= S_1
$M(S_1, b)$: {1, 2, 4, 5, 6, 7, 8, 9, 10} = S_2
$M(S_2, a)$: {1, 2, 3, 4, 6, 7, 8, 11, 12} = S_3
$M(S_2, b)$: {1, 2, 4, 5, 6, 7, 8, 9, 10} = S_2
$M(S_3, a)$: {1, 2, 3, 4, 6, 7, 8, 13} = S_4
$M(S_3, b)$: {1, 2, 4, 5, 6, 7, 8, 9, 10} = S_2
$M(S_4, a)$: {1, 2, 3, 4, 6, 7, 8} = S_1
$M(S_4, b)$: {1, 2, 4, 5, 6, 7, 8, 9, 10} = S_2

The DFA itself is illustrated in Figure 2.22.

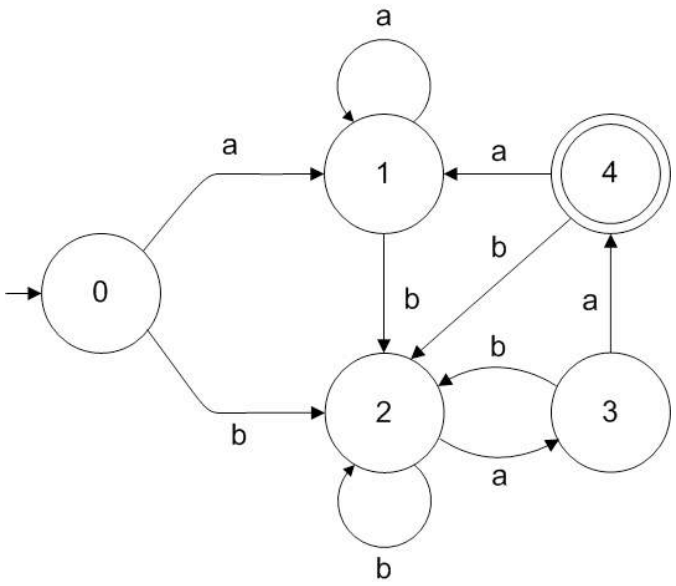


Figure 2.22 A DFA recognizing (a|b)*baa

Finally, we use partitioning to produce the minimal DFA illustrated in Figure 2.24.

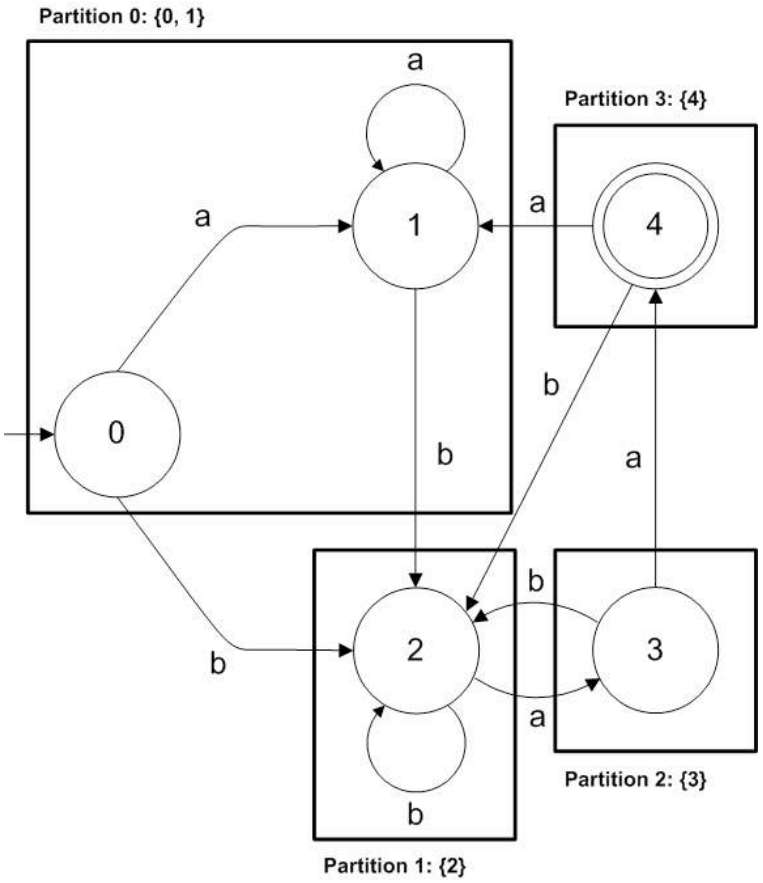


Figure 2.23 Partition of 2.22

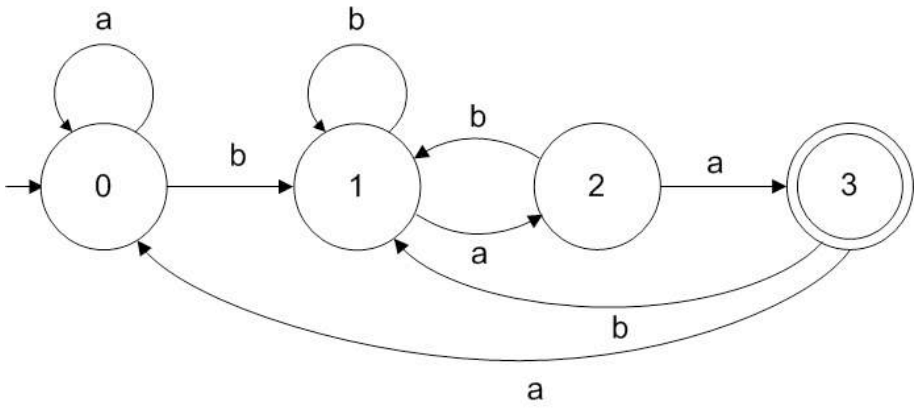


Figure 2.24 minimal DFA recognizing (a|b)*baa

2.9 JavaCC: a Tool for Generating Scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating lexical analyzers from regular expressions, and parsers from context-free grammars. In this section we are interested in the former; we visit the latter in the next chapter.

A lexical grammar specification takes the form of a set of regular expressions and a set of states; from any particular state, only certain regular expressions may be matched in scanning the input. There is a standard DEFAULT state, in which scanning generally begins. One may specify additional states as required.

Scanning a token proceeds by considering all regular expressions in the current state and choosing that which consumes the greatest number of input characters. After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state.

There are four *kinds* of regular expressions, determining what happens when the regular expression has been matched:

1. SKIP: throws away the matched string.
2. MORE: continues to the next state, taking the matched string along.
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller).
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing.

Examples

For example, a SKIP can be used for ignoring white space:

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

This matches one of the white space characters and throws it away; because we do not specify a next state, the scanner remains in the current (DEFAULT) state. We can deal with single-line comments with the following regular expressions:

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT>
MORE: { < ~[] > }
```

Matching the `//` puts the scanner into the `IN_SINGLE_LINE_COMMENT` state. The next two regular expressions apply only to this state. The first matches an end of line and returns it as a special token (which is not seen by the parser); it then puts the scanner back into the `DEFAULT` state. The second matches anything else and throws it away; because no next

state is specified, the scanner remains in the **IN_SINGLE_LINE_COMMENT** state. An alternative regular expression dealing with single-line comments is simpler[4],

```
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

One may easily specify the syntax of reserved words and symbols by spelling them out, e.g.,

```
TOKEN:
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
…

| < COMMA: "," >
| < DOT: "." >
}
```

The Java identifier preceding the colon, e.g., **ABSTRACT**, **BOOLEAN**, **COMMA**, and **DOT**, represents the token's kind. Each token also has an *image* that holds onto the actual input string that matches the regular expression following the colon.

A more interesting token is that for scanning identifiers:

```
TOKEN:
{ < IDENTIFIER: (<LETTER>|"_"|"$") (<LETTER>|<DIGIT>|"_"|"$")* >
| < #LETTER: ["a"-"z","A"-"Z"] >
| < #DIGIT: ["0"-"9"] >
}
```

This says that an **IDENTIFIER** is a letter, underscore or dollar sign, followed by zero or more letters, digits, underscores and dollar signs. Here the image records the identifier itself. The # preceding **LETTER** and **DIGIT** indicates that these two identifiers are private to the scanner and so unknown to the parser.

Literals are also relatively straightforward:

```
TOKEN:
{ < INT_LITERAL: ("0" | <NON_ZERO_DIGIT> (<DIGIT>)*) >
| < #NON_ZERO_DIGIT: ["1"-"9"] >
| < CHAR_LITERAL: "'" (<ESC> | ~["'","\\","\n","\r"]) "'" >
| < STRING_LITERAL: "\"" (<ESC> | ~["\"","\\","\n","\r"])* "\"" >
| < #ESC: "\\" ["n","t","b","r","f","\\","'","\""] >
```

[4] Both implementations of the single-line comment come from the examples and documentation distributed with JavaCC. This simpler one comes from the TokenManager Mini-tutorial at https://javacc.dev.java.net/doc/tokenmanager.html.

```
}
```

JavaCC takes a specification of the lexical syntax and produces several Java files. One of these, **`TokenManager.java`**, defines a program that implements a state machine; this is our scanner.

To see the entire lexical grammar for *j--*, read the JavaCC input file, j--.jj, in the jminusminus package; the lexical grammar is close to the top of that file.

Further Reading

The lexical syntax for Java may be found in James (Gosling et al, 2005); this book is also published online on the Sun (but now Oracle) Developer Network at http://java.sun.com/docs/books/jls.

For a more rigorous presentation of finite state automata and their proofs, see (Sipser, 2005) or (Linz, 2006). There is also the classic (Hopcroft and Ullman, 1969).

JavaCC is distributed with both documentation and examples; see https://javacc.dev.java.net. Also see (Copeland, 2007) for a nice guide to using JavaCC.

Lex is a classic lexical analyzer generator for the C programming language. The best description of its use is still (Lesk, 1975). See (Lesk and Schmidt) for an implementation of Lex at http://dinosaur.compilertools.net/lex/index.html. An open-source implementation called Flex, originally written by Vern Paxton and called Flex, may be found at http://flex.sourceforge.net.

Exercises

2.1 Consult Chapter 3 (Lexical Structure) of *The Java Language Specification, Third Edition* (see Further Reading, above). There you will find a complete specification of Java's lexical syntax.

a. Make a list of all the keywords that are in Java but not in *j--*.
b. Make a list of the escape sequences that are in Java but are not in *j--*.
c. How do Java identifiers differ from *j--* identifiers?
d. How do Java integer literals differ from *j--* integer literals?

2.2 Draw the state transition diagram that recognizes Java multi-line comments, beginning with a `/*` and ending with `*/`.

2.3 Draw the state transition diagram for recognizing Java integer literals.

2.4 Write a regular expression that describes the language of all Java integer literals.

2.5 Draw the state transition diagram that recognizes all Java numerical literals (both integers and floating point).

2.6 Write a regular expression that describes all Java numeric literals (both integers and floating point).

2.7 For each of the following regular expressions, use Thompson's method to derive a non-deterministic finite automaton (NFA) recognizing the same language.

a. aaa
b. (ab)*ab
c. a*bc*d
d. (a|bc*)a*
e. (a|b)*
f. a*|b*
g. (a*|b*)*
h. ((aa)*(ab)*(ba)*(bb)*)*

2.8 For each of the NFA's in the previous exercise, use powerset construction for deriving an equivalent deterministic finite automaton (DFA).

2.9 For each of the DFA's in the previous exercise, use the partitioning method to derive an equivalent, minimal DFA.

The following exercises ask you to modify the handcrafted scanner in the *j--* compiler for recognizing new categories of tokens. For each of these, write a suitable set of tests, then add the necessary code, and run the tests.

2.10 Modify **`Scanner`** in the *j--* compiler to scan (and ignore) *Java* multi-line comments.

2.11 Modify **`Scanner`** in the *j--* compiler to recognize and return all *Java* operators.

2.12 Modify **`Scanner`** in the *j--* compiler to recognize and return all *Java* reserved words.

2.13 Modify **`Scanner`** in the *j--* compiler to recognize and return all *Java* double precision literals (returned as **`DOUBLE_LITERAL`**).

2.14 Modify **`Scanner`** in the *j--* compiler to recognize and return all other literals in Java, e.g. **`FLOAT_LITERAL`**, **`LONG_LITERAL`**, etc.

2.15 Modify **`Scanner`** in the *j--* compiler to recognize and return all other representations of integers (hexadecimal, octal, etc).

The following exercises ask you to modify the **`j--.jj`** file in the *j--* compiler for recognizing new categories of tokens. For each of these, write a suitable set of tests, then add the necessary code, and run the tests. Consult Appendix A to learn how tests work.

2.16 Modify the **`j--.jj`** file in the *j--* compiler to scan (and ignore) *Java* multi-line comments.

2.17 Modify the **`j--.jj`** file in the *j--* compiler to recognize and return all *Java* operators.

2.18 Modify the **`j--.jj`** file in the *j--* compiler to recognize and return all *Java* reserved words.

2.19 Modify the **`j--.jj`** file in the *j--* compiler to recognize and return all *Java* double precision literals (returned as **`DOUBLE_LITERAL`**).

2.20 Modify the **`j--.jj`** file in the *j--* compiler to recognize and return all other literals in Java, e.g. **`FLOAT_LITERAL`**, **`LONG_LITERAL`**, etc.

2.21 Modify the **`j--.jj`** file in the *j--* compiler to recognize and return all other representations of integers (hexadecimal, octal, etc).