2

Lexical Analysis

lex-i-cal: of or relating to words or the vocabulary of a language as distinguished from its grammar and construction

Webster's Dictionary

To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way. The front end of the compiler performs analysis; the back end does synthesis.

The analysis is usually broken up into

Lexical analysis: breaking the input into individual words or "tokens";
Syntax analysis: parsing the phrase structure of the program; and
Semantic analysis: calculating the program's meaning.

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens. It would unduly complicate the parser to have to account for possible white space and comments at every possible point; this is the main reason for separating lexical analysis from parsing.

Lexical analysis is not very complicated, but we will attack it with high-powered formalisms and tools, because similar formalisms will be useful in the study of parsing and similar tools have many applications in areas other than compilation.

2.1 LEXICAL TOKENS

A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language. A programming language classifies lexical tokens into a finite set of token types. For example, some of the token types of a typical programming language are:

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`
RPAREN	`)`

Punctuation tokens such as IF, VOID, RETURN constructed from alphabetic characters are called *reserved words* and, in most languages, cannot be used as identifiers.

Examples of nontokens are

comment	`/* try again */`
preprocessor directive	`#include<stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, and newlines	

In languages weak enough to require a macro preprocessor, the preprocessor operates on the source character stream, producing another character stream that is then fed to the lexical analyzer. It is also possible to integrate macro processing with lexical analysis.

Given a program such as

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

the lexical analyzer will return the stream

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s)

```
COMMA    STRING(0.0)    COMMA    NUM(3)    RPAREN    RPAREN
RETURN   REAL(0.0)    SEMI    RBRACE    EOF
```

where the token-type of each token is reported; some of the tokens, such as identifiers and literals, have *semantic values* attached to them, giving auxiliary information in addition to the token type.

How should the lexical rules of a programming language be described? In what language should a lexical analyzer be written?

We can describe the lexical tokens of a language in English; here is a description of identifiers in C or Java:

> An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

And any reasonable programming language serves to implement an ad hoc lexer. But we will specify lexical tokens using the formal language of *regular expressions*, implement lexers using *deterministic finite automata*, and use mathematics to connect the two. This will lead to simpler and more readable lexical analyzers.

2.2 REGULAR EXPRESSIONS

Let us say that a *language* is a set of *strings*; a string is a finite sequence of *symbols*. The symbols themselves are taken from a finite *alphabet*.

The Pascal language is the set of all strings that constitute legal Pascal programs; the language of primes is the set of all decimal-digit strings that represent prime numbers; and the language of C reserved words is the set of all alphabetic strings that cannot be used as identifiers in the C programming language. The first two of these languages are infinite sets; the last is a finite set. In all of these cases, the alphabet is the ASCII character set.

When we speak of languages in this way, we will not assign any meaning to the strings; we will just be attempting to classify each string as in the language or not.

To specify some of these (possibly infinite) languages with finite descrip-

tions, we will use the notation of *regular expressions*. Each regular expression stands for a set of strings.

Symbol: For each symbol **a** in the alphabet of the language, the regular expression **a** denotes the language containing just the string `a`.

Alternation: Given two regular expressions M and N, the alternation operator written as a vertical bar | makes a new regular expression $M \mid N$. A string is in the language of $M \mid N$ if it is in the language of M or in the language of N. Thus, the language of **a** | **b** contains the two strings `a` and `b`.

Concatenation: Given two regular expressions M and N, the concatenation operator · makes a new regular expression $M \cdot N$. A string is in the language of $M \cdot N$ if it is the concatenation of any two strings α and β such that α is in the language of M and β is in the language of N. Thus, the regular expression (**a** | **b**) · **a** defines the language containing the two strings `aa` and `ba`.

Epsilon: The regular expression ϵ represents a language whose only string is the empty string. Thus, $(a \cdot b) \mid \epsilon$ represents the language {`""`,`"ab"`}.

Repetition: Given a regular expression M, its Kleene closure is M^*. A string is in M^* if it is the concatenation of zero or more strings, all of which are in M. Thus, ((**a** | **b**) · **a**)* represents the infinite set { `""`, `"aa"`, `"ba"`, `"aaaa"`, `"baaa"`, `"aaba"`, `"baba"`, `"aaaaaa"`, ... }.

Using symbols, alternation, concatenation, epsilon, and Kleene closure we can specify the set of ASCII characters corresponding to the lexical tokens of a programming language. First, consider some examples:

(0 | 1)* · 0	Binary numbers that are multiples of two.
b*(abb*)*(a|ϵ)	Strings of `a`'s and `b`'s with no consecutive `a`'s.
(a|b)*aa(a|b)*	Strings of `a`'s and `b`'s containing consecutive `a`'s.

In writing regular expressions, we will sometimes omit the concatenation symbol or the epsilon, and we will assume that Kleene closure "binds tighter" than concatenation, and concatenation binds tighter than alternation; so that **ab** | **c** means (**a** · **b**) | **c**, and (**a** |) means (**a** | ϵ).

Let us introduce some more abbreviations: **[abcd]** means (**a** | **b** | **c** | **d**), **[b-g]** means **[bcdefg]**, **[b-gM-Qkr]** means **[bcdefgMNOPQkr]**, $M?$ means $(M \mid \epsilon)$, and M^+ means $(M \cdot M^*)$. These extensions are convenient, but none extend the descriptive power of regular expressions: Any set of strings that can be described with these abbreviations could also be described by just the basic set of operators. All the operators are summarized in Figure 2.1.

Using this language, we can specify the lexical tokens of a programming language (Figure 2.2). For each token, we supply a fragment of ML code that reports which token type has been recognized.

a	An ordinary character stands for itself.
ϵ	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to write concatenation.
M^*	Repetition (zero or more times).
M^+	Repetition, one or more times.
$M?$	Optional, zero or one occurrence of M.
[a – zA – Z]	Character set alternation.
.	A period stands for any single character except newline.
`"a.+*"`	Quotation, a string in quotes stands for itself literally.

FIGURE 2.1. Regular expression notation.

```
if                                          (IF);
[a-z][a-z0-9]*                              (ID);
[0-9]+                                      (NUM);
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)         (REAL);
("--"[a-z]*"\n")|(" "|"\n"|"\t")+           (continue());
.                                           (error(); continue());
```

FIGURE 2.2. Regular expressions for some tokens.

The fifth line of the description recognizes comments or white space, but does not report back to the parser. Instead, the white space is discarded and the lexer resumed; this is what `continue` does. The comments for this lexer begin with two dashes, contain only alphabetic characters, and end with newline.

Finally, a lexical specification should be *complete*, always matching some initial substring of the input; we can always achieve this by having a rule that matches any single character (and in this case, prints an "illegal character" error message and continues).

These rules are a bit ambiguous. For example, does `if8` match as a single identifier or as the two tokens `if` and `8`? Does the string `if 89` begin with an identifier or a reserved word? There are two important disambiguation rules used by Lex, ML-Lex, and other similar lexical-analyzer generators:

Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.

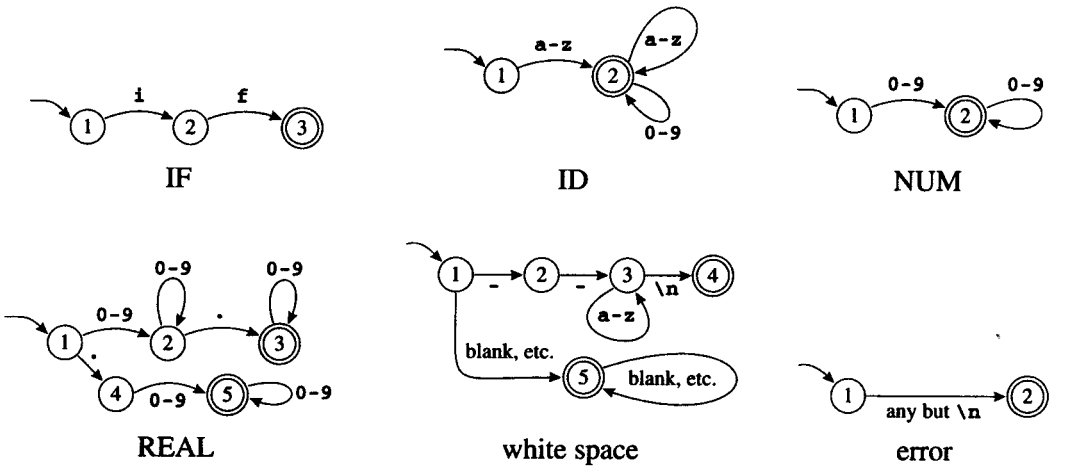


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

Rule priority: For a *particular* longest initial substring, the first regular expression that can match determines its token type. This means that the order of writing down the regular-expression rules has significance.

Thus, `if8` matches as an identifier by the longest-match rule, and `if` matches as a reserved word by rule-priority.

2.3 FINITE AUTOMATA

Regular expressions are convenient for specifying lexical tokens, but we need a formalism that can be implemented as a computer program. For this we can use finite automata (N.B. the singular of automata is automaton). A finite automaton has a finite set of *states*; *edges* lead from one state to another, and each edge is labeled with a *symbol*. One state is the *start* state, and certain of the states are distinguished as *final* states.

Figure 2.3 shows some finite automata. We number the states just for convenience in discussion. The start state is numbered 1 in each case. An edge labeled with several characters is shorthand for many parallel edges; so in the ID machine there are really 26 edges each leading from state 1 to 2, each labeled by a different letter.

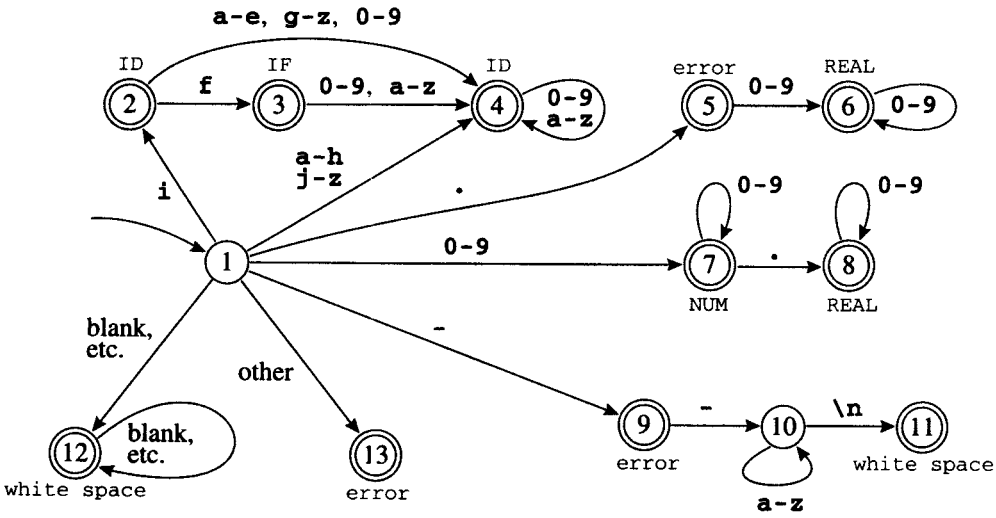


FIGURE 2.4. Combined finite automaton.

In a *deterministic* finite automaton (DFA), no two edges leaving from the same state are labeled with the same symbol. A DFA *accepts* or *rejects* a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character. After making n transitions for an n-character string, if the automaton is in a final state, then it accepts the string. If it is not in a final state, or if at some point there was no appropriately labeled edge to follow, it rejects. The *language* recognized by an automaton is the set of strings that it accepts.

For example, it is clear that any string in the language recognized by automaton ID must begin with a letter. Any single letter leads to state 2, which is final; so a single-letter string is accepted. From state 2, any letter or digit leads back to state 2, so a letter followed by any number of letters and digits is also accepted.

In fact, the machines shown in Figure 2.3 accept the same languages as the regular expressions of Figure 2.2.

These are six separate automata; how can they be combined into a single machine that can serve as a lexical analyzer? We will study formal ways of doing this in the next section, but here we will just do it ad hoc: Figure 2.4 shows such a machine. Each final state must be labeled with the token-type

that it accepts. State 2 in this machine has aspects of state 2 of the IF machine and state 2 of the ID machine; since the latter is final, then the combined state must be final. State 3 is like state 3 of the IF machine and state 2 of the ID machine; because these are both final we use *rule priority* to disambiguate – we label state 3 with IF because we want this token to be recognized as a reserved word, not an identifier.

We can encode this machine as a transition matrix: a two-dimensional array (a vector of vectors), subscripted by state number and input character. There will be a "dead" state (state 0) that loops to itself on all characters; we use this to encode the absence of an edge.

```
val edges =
 vector[          (*       ···0 1 2···-···e f g h i j···*)
(* state 0 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 1 *) vector[0,0,···7,7,7···9···4,4,4,4,2,4···],
(* state 2 *) vector[0,0,···4,4,4···0···4,3,4,4,4,4···],
(* state 3 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 4 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 5 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 6 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 7 *) vector[0,0,···7,7,7···0···0,0,0,0,0,0···],
   et cetera
        ]
```

There must also be a "finality" array, mapping state numbers to actions – final state 2 maps to action ID, and so on.

RECOGNIZING THE LONGEST MATCH

It is easy to see how to use this table to recognize whether to accept or reject a string, but the job of a lexical analyzer is to find the longest match, the longest initial substring of the input that is a valid token. While interpreting transitions, the lexer must keep track of the longest match seen so far, and the position of that match.

Keeping track of the longest match just means remembering the last time the automaton was in a final state with two variables, `Last-Final` (the state number of the most recent final state encountered) and `Input-Position-at-Last-Final`. Every time a final state is entered, the lexer updates these variables; when a *dead* state (a nonfinal state with no output transitions) is reached, the variables tell what token was matched, and where it ended.

Figure 2.5 shows the operation of a lexical analyzer that recognizes longest matches; note that the current input position may be far beyond the most recent position at which the recognizer was in a final state.

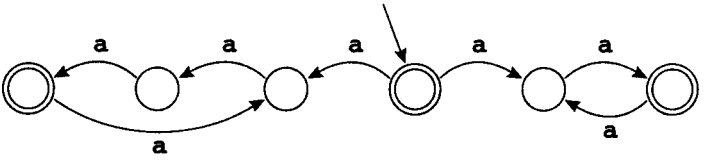
Last Final	Current State	Current Input	Accept Action
0	1	⊥if --not-a-com	
2	2	|i⊥f --not-a-com	
3	3	|if⊥ --not-a-com	
3	0	|if⊤⊥--not-a-com	*return* IF
0	1	if⊥ --not-a-com	
12	12	if| ⊥--not-a-com	
12	0	if| ⊤⊥not-a-com	*found white space; resume*
0	1	if ⊥--not-a-com	
9	9	if |-⊥not-a-com	
9	10	if |-⊤⊥ot-a-com	
9	10	if |-⊤n⊥t-a-com	
9	10	if |-⊤no⊥-a-com	
9	10	if |-⊤not⊥a-com	
9	0	if |-⊤not-⊥-com	*error, illegal token '-'; resume*
0	1	if -⊥not-a-com	
9	9	if -|-⊥ot-a-com	
9	0	if -|-⊤⊥ot-a-com	*error, illegal token '-'; resume*

FIGURE 2.5. The automaton of Figure 2.4 recognizes several tokens. The symbol | indicates the input position at each successive call to the lexical analyzer, the symbol ⊥ indicates the current position of the automaton, and ⊤ indicates the most recent position in which the recognizer was in a final state.

2.4 NONDETERMINISTIC FINITE AUTOMATA

A nondeterministic finite automaton (NFA) is one that has a choice of edges – labeled with the same symbol – to follow out of a state. Or it may have special edges labeled with ϵ (the Greek letter epsilon), that can be followed without eating any symbol from the input.

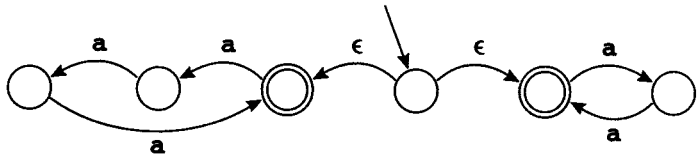
Here is an example of an NFA:



In the start state, on input character a, the automaton can move either right or left. If left is chosen, then strings of a's whose length is a multiple of three will be accepted. If right is chosen, then even-length strings will be accepted. Thus, the language recognized by this NFA is the set of all strings of a's whose length is a multiple of two or three.

On the first transition, this machine must choose which way to go. It is required to accept the string if there is *any* choice of paths that will lead to acceptance. Thus, it must "guess," and must always guess correctly.

Edges labeled with ϵ may be taken without using up a symbol from the input. Here is another NFA that accepts the same language:

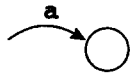


Again, the machine must choose which ϵ-edge to take. If there is a state with some ϵ-edges and some edges labeled by symbols, the machine can choose to eat an input symbol (and follow the corresponding symbol-labeled edge), or to follow an ϵ-edge instead.

CONVERTING A REGULAR EXPRESSION TO AN NFA

Nondeterministic automata are a useful notion because it is easy to convert a (static, declarative) regular expression to a (simulatable, quasi-executable) NFA.

The conversion algorithm turns each regular expression into an NFA with a *tail* (start edge) and a *head* (ending state). For example, the single-symbol regular expression **a** converts to the NFA



The regular expression **ab**, made by combining **a** with **b** using concatenation is made by combining the two NFAs, hooking the head of **a** to the tail of **b**. The resulting machine has a tail labeled by **a** and a head into which the **b** edge flows.

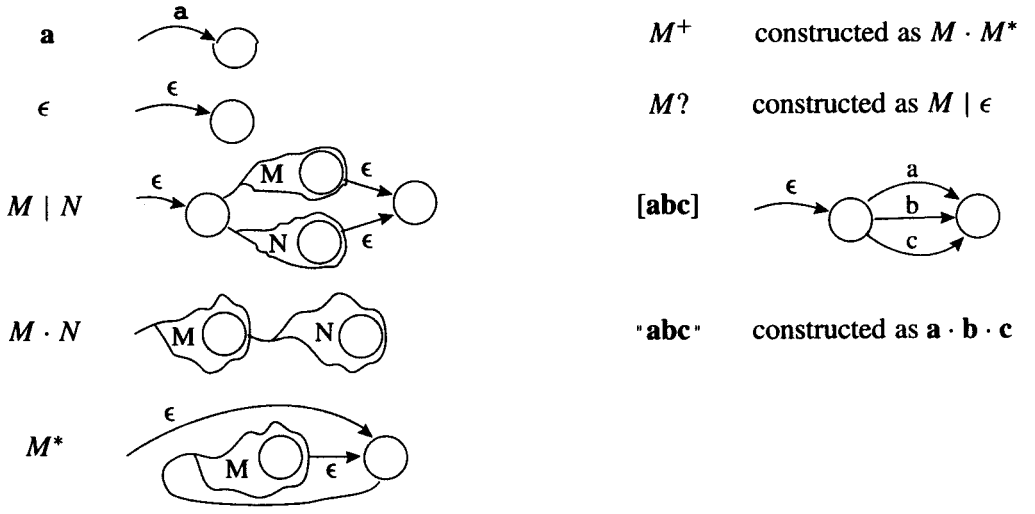


FIGURE 2.6. Translation of regular expressions to NFAs.

In general, any regular expression M will have some NFA with a tail and head:

We can define the translation of regular expressions to NFAs by induction. Either an expression is primitive (a single symbol or ϵ) or it is made from smaller expressions. Similarly, the NFA will be primitive or made from smaller NFAs.

Figure 2.6 shows the rules for translating regular expressions to nondeterministic automata. We illustrate the algorithm on some of the expressions in Figure 2.2 – for the tokens IF, ID, NUM, and **error**. Each expression is translated to an NFA, the "head" state of each NFA is marked final with a different token type, and the tails of all the expressions are joined to a new start node. The result – after some merging of equivalent NFA states – is shown in Figure 2.7.

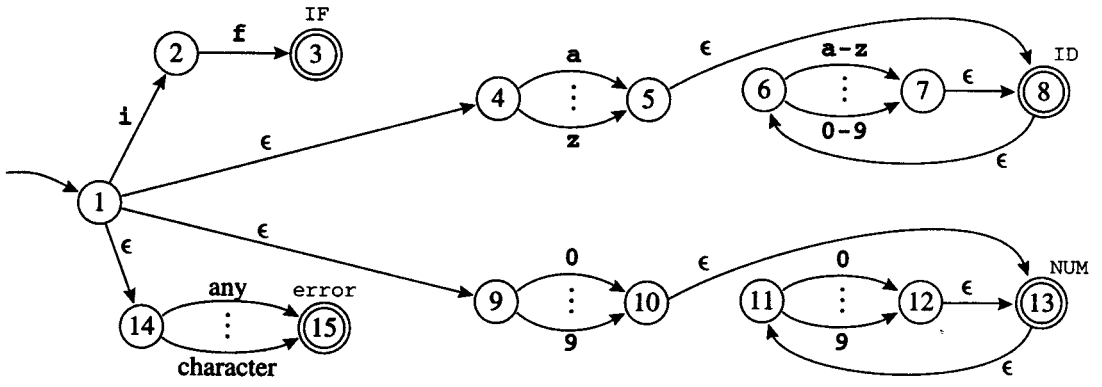


FIGURE 2.7. Four regular expressions translated to an NFA.

CONVERTING AN NFA TO A DFA

As we saw in Section 2.3, implementing deterministic finite automata (DFAs) as computer programs is easy. But implementing NFAs is a bit harder, since most computers don't have good "guessing" hardware.

We can avoid the need to guess by trying every possibility at once. Let us simulate the NFA of Figure 2.7 on the string `in`. We start in state 1. Now, instead of guessing which ϵ-transition to take, we just say that at this point the NFA might take any of them, so it is in one of the states {1, 4, 9, 14}; that is, we compute the *ϵ-closure* of {1}. Clearly, there are no other states reachable without eating the first character of the input.

Now, we make the transition on the character `i`. From state 1 we can reach 2, from 4 we reach 5, from 9 we go nowhere, and from 14 we reach 15. So we have the set {2, 5, 15}. But again we must compute ϵ-closure: from 5 there is an ϵ-transition to 8, and from 8 to 6. So the NFA must be in one of the states {2, 5, 6, 8, 15}.

On the character `n`, we get from state 6 to 7, from 2 to nowhere, from 5 to nowhere, from 8 to nowhere, and from 15 to nowhere. So we have the set {7}; its ϵ-closure is {6, 7, 8}.

Now we are at the end of the string `in`; is the NFA in a final state? One of the states in our possible-states set is 8, which is final. Thus, `in` is an ID token.

We formally define ϵ-closure as follows. Let **edge**(s, c) be the set of all NFA states reachable by following a single edge with label c from state s.

For a set of states S, **closure**(S) is the set of states that can be reached from a state in S without consuming any of the input, that is, by going only through ϵ edges. Mathematically, we can express the idea of going through ϵ edges by saying that **closure**(S) is smallest set T such that

$$T = S \cup \left(\bigcup_{s \in T} \textbf{edge}(s, \epsilon) \right).$$

We can calculate T by iteration:

$T \leftarrow S$
repeat $T' \leftarrow T$
$\quad T \leftarrow T' \cup (\bigcup_{s \in T'} \textbf{edge}(s, \epsilon))$
until $T = T'$

Why does this algorithm work? T can only grow in each iteration, so the final T must include S. If $T = T'$ after an iteration step, then T must also include $\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)$. Finally, the algorithm must terminate, because there are only a finite number of distinct states in the NFA.

Now, when simulating an NFA as described above, suppose we are in a set $d = \{s_i, s_k, s_l\}$ of NFA states s_i, s_k, s_l. By starting in d and eating the input symbol c, we reach a new set of NFA states; we'll call this set **DFAedge**(d, c):

$$\textbf{DFAedge}(d, c) = \textbf{closure}(\bigcup_{s \in d} \textbf{edge}(s, c))$$

Using **DFAedge**, we can write the NFA simulation algorithm more formally. If the start state of the NFA is s_1, and the input string is $c_1, \ldots, c_k$, then the algorithm is:

$d \leftarrow$ **closure**$(\{s_1\})$
for $i \leftarrow 1$ **to** k
$\quad d \leftarrow$ **DFAedge**(d, c_i)

Manipulating sets of states is expensive – too costly to want to do on every character in the source program that is being lexically analyzed. But it is possible to do all the sets-of-states calculations in advance. We make a DFA from the NFA, such that each set of NFA states corresponds to one DFA state. Since the NFA has a finite number n of states, the DFA will also have a finite number (at most 2^n) of states.

DFA construction is easy once we have **closure** and **DFAedge** algorithms. The DFA start state d_1 is just **closure**(s_1), as in the NFA simulation algo-

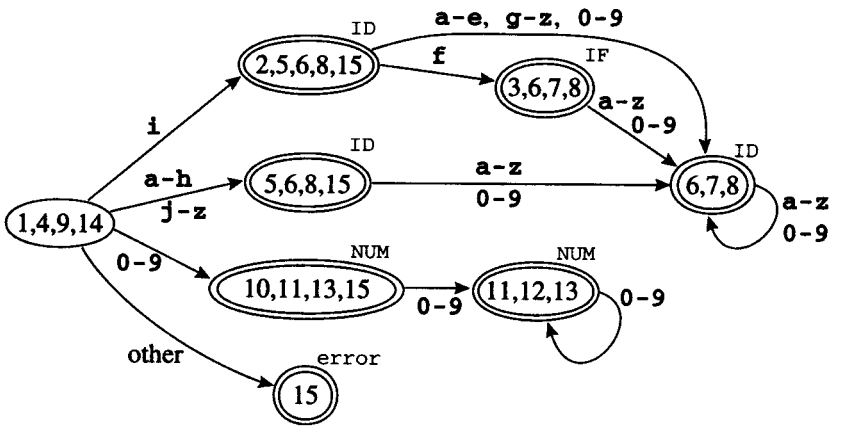


FIGURE 2.8. NFA converted to DFA.

rithm. Abstractly, there is an edge from d_i to d_j labeled with c if d_j = **DFAedge**(d_i, c). We let Σ be the alphabet.

```
states[0] ← {};     states[1] ← closure({s1})
p ← 1;     j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```

The algorithm does not visit unreachable states of the DFA. This is extremely important, because in principle the DFA has 2^n states, but in practice we usually find that only about n of them are reachable from the start state. It is important to avoid an exponential blowup in the size of the DFA interpreter's transition tables, which will form part of the working compiler.

A state d is *final* in the DFA if any NFA-state in states[d] is final in the NFA. Labeling a state *final* is not enough; we must also say what token is recognized; and perhaps several members of states[d] are final in the NFA. In this case we label d with the token-type that occurred first in the list of

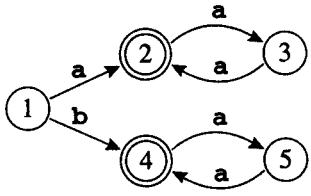
regular expressions that constitute the lexical specification. This is how *rule priority* is implemented.

After the DFA is constructed, the "states" array may be discarded, and the "trans" array is used for lexical analysis.

Applying the DFA construction algorithm to the NFA of Figure 2.7 gives the automaton in Figure 2.8.

This automaton is suboptimal. That is, it is not the smallest one that recognizes the same language. In general, we say that two states s_1 and s_2 are equivalent when the machine starting in s_1 accepts a string σ if and only if starting in s_2 it accepts σ. This is certainly true of the states labeled 5,6,8,15 and 6,7,8 in Figure 2.8; and of the states labeled 10,11,13,15 and 11,12,13. In an automaton with two equivalent states s_1 and s_2, we can make all of s_2's incoming edges point to s_1 instead and delete s_2.

How can we find equivalent states? Certainly, s_1 and s_2 are equivalent if they are both final or both non-final and for any symbol c, trans$[s_1, c]$ = trans$[s_2, c]$; 10,11,13,15 and 11,12,13 satisfy this criterion. But this condition is not sufficiently general; consider the automaton



Here, states 2 and 4 are equivalent, but trans$[2, a] \neq$ trans$[4, a]$.

After constructing a DFA it is useful to apply an algorithm to minimize it by finding equivalent states; see Exercise 2.6.

2.5 ML-Lex: A LEXICAL ANALYZER GENERATOR

DFA construction is a mechanical task easily performed by computer, so it makes sense to have an automatic *lexical analyzer generator* to translate regular expressions into a DFA.

ML-Lex is a lexical analyzer generator that produces an ML program from a *lexical specification*. For each token type in the programming language to be lexically analyzed, the specification contains a regular expression and an

```
(* ML Declarations: *)
    type lexresult = Tokens.token
    fun eof() = Tokens.EOF(0,0)
    %%
(* Lex Definitions: *)
    digits=[0-9]+
    %%
(* Regular Expressions and Actions: *)
    if                  => (Tokens.IF(yypos,yypos+2));
    [a-z][a-z0-9]*      => (Tokens.ID(yytext,yypos,yypos+size yytext));
    {digits}            => (Tokens.NUM(Int.fromString yytext,
                                yypos,yypos+size yytext));
    ({digits}"."[0-9]*)|([0-9]*"."{digits})
                        => (Tokens.REAL(Real.fromString yytext,
                                    yypos, yypos+size yytext));
    ("--"[a-z]*"\n")|(" "|"\n"|"\t")+
                        =>     (continue());
                        =>     (ErrorMsg.error yypos "illegal character";
                                continue());
```

PROGRAM 2.9. ML-Lex specification of the tokens from Figure 2.2.

action. The action communicates the token type (perhaps along with other information) to the next phase of the compiler.

The output of ML-Lex is a program in ML – a lexical analyzer that interprets a DFA using the algorithm described in Section 2.3 and executes the action fragments on each match. The action fragments are just ML statements that return token values.

The tokens described in Figure 2.2 are specified in ML-Lex as shown in Program 2.9.

The first part of the specification, above the first `%%` mark, contains functions and types written in ML. These must include the type `lexresult`, which is the result type of each call to the lexing function; and the function `eof`, which the lexing engine will call at end of file. This section can also contain utility functions for the use of the semantic actions in the third section.

The second part of the specification contains regular-expression abbreviations and state declarations. For example, the declaration `digits=[0-9]+` in this section allows the name `{digits}` to stand for a nonempty sequence of digits within regular expressions.

The third part contains regular expressions and actions. The actions are fragments of ordinary ML code. Each action must return a value of type

`lexresult`. In this specification, `lexresult` is a token from the `Tokens` structure.

In the action fragments, several special variables are available. The string matched by the regular expression is `yytext`. The file position of the beginning of the matched string is `yypos`. The function `continue()` calls the lexical analyzer recursively.

In this particular example, each token is a data constructor parameterized by two integers indicating the position – in the input file – of the beginning and end of the token.

```
structure Tokens =
struct
   type pos = int
   datatype token = EOF of pos * pos
                  | IF of pos * pos
                  | ID of string * pos * pos
                  | NUM of int * pos * pos
                  | REAL of real * pos * pos
                    ⋮
end
```

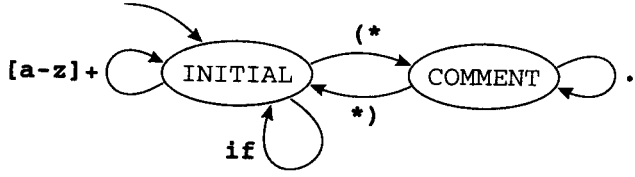
Thus, it is appropriate to pass `yypos` and `yypos + size(yytext)` to the constructor. Some tokens have *semantic values* associated with them. For example, ID's semantic value is the character string constituting the identifier; NUM's semantic value is an integer; and IF has no semantic value (one IF is indistinguishable from another). Thus, the ID and NUM constructors have an extra argument for the semantic value, and this value can be computed from `yytext`.

START STATES

Regular expressions are *static* and *declarative*; automata are *dynamic* and *imperative*. That is, you can see the components and structure of a regular expression without having to simulate an algorithm, but to understand an automaton it is often necessary to "execute" it in your mind. Thus, regular expressions are usually more convenient to specify the lexical structure of programming-language tokens.

But sometimes the step-by-step, state-transition model of automata is appropriate. ML-Lex has a mechanism to mix states with regular expressions. One can declare a set of *start states*; each regular expression can be prefixed by the set of start states in which it is valid. The action fragments can explicitly change the start state. In effect, we have a finite automaton whose edges

are labeled, not by single symbols, but by regular expressions. This example shows a language with simple identifiers, `if` tokens, and comments delimited by `(*` and `*)` brackets:



Though it is possible to write a single regular expression that matches an entire comment, as comments get more complicated it becomes more difficult, or even impossible if nested comments are allowed.

The ML-Lex specification corresponding to this machine is

```
  ⋮  the usual preamble ...
%%
%s COMMENT
%%
<INITIAL>if          => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+      => (Tokens.ID(yytext,yypos,
                                   yypos+size(yytext)));
<INITIAL>"(*"        => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"        => (YYBEGIN INITIAL; continue());
<COMMENT>.           => (continue();)
```

where INITIAL is the start state, provided by default in all specifications. Any regular expression not prefixed by a `<STATE>` operates in all states; this feature is rarely useful.

This example can be easily augmented to handle nested comments, via a global variable that is incremented and decremented in the semantic actions.

PROGRAM LEXICAL ANALYSIS

Use ML-Lex to implement a lexical analyzer for the Tiger language. Appendix A describes, among other things, the lexical tokens of Tiger.

This chapter has left out some of the specifics of how the lexical analyzer should be initialized and how it should communicate with the rest of the compiler. You can learn this from the ML-Lex manual, but the "skeleton" files in the `$TIGER/chap2` directory will also help get you started.

Along with the `tiger.lex` file you should turn in documentation for the following points:

- how you handle comments;
- how you handle strings;
- error handling;
- end-of-file handling;
- other interesting features of your lexer.

Supporting files are available in `$TIGER/chap2` as follows:

`tokens.sig` Signature of the `Tokens` structure.
`tokens.sml` The `Tokens` structure, containing the `token` type and constructors that your lexer should use to build instances of the `token` type. It is important to do it in this way, so that things will still work when the "real" parser is attached to this lexer, and the "real" `Tokens` structure is used.
`errormsg.sml` The `ErrorMsg` structure, useful for producing error messages with file names and line numbers.
`driver.sml` A test scaffold to run your lexer on an input file.
`tiger.lex` The beginnings of a real `tiger.lex` file.
`sources.cm` A "makefile" for the ML Compilation Manager.

When reading the *Tiger Language Reference Manual* (Appendix A), pay particular attention to the paragraphs with the headings **Identifiers, Comments, Integer literal,** and **String literal**.

The reserved words of the language are: `while`, `for`, `to`, `break`, `let`, `in`, `end`, `function`, `var`, `type`, `array`, `if`, `then`, `else`, `do`, `of`, `nil`.

The punctuation symbols used in the language are:

```
, : ; ( ) [ ] { } . + - * / = <> < <= > >= & | :=
```

The string value that you return for a string literal should have all the escape sequences translated into their meanings.

There are no negative integer literals; return two separate tokens for `-32`.

Detect unclosed comments (at end of file) and unclosed strings.

The directory `$TIGER/testcases` contains a few sample Tiger programs.

To get started: Make a directory and copy the contents of `$TIGER/chap2` into it. Make a file `test.tig` containing a short program in the Tiger language. Then execute `sml` and type the command `CM.make();` the CM (compilation manager) *make* system will run `ml-lex`, if needed, and will compile and link the ML source files (as needed).

Finally, `Parse.parse "test.tig";` will lexically analyze the file using a test scaffold.

FURTHER READING

Lex was the first lexical-analyzer generator based on regular expressions [Lesk 1975]; it is still widely used.

Computing ϵ-closure can be done more efficiently by keeping a queue or stack of states whose edges have not yet been checked for ϵ-transitions [Aho et al. 1986]. Regular expressions can be converted directly to DFAs without going through NFAs [McNaughton and Yamada 1960; Aho et al. 1986].

DFA transition tables can be very large and sparse. If represented as a simple two-dimensional matrix (*states* × *symbols*) they take far too much memory. In practice, tables are compressed; this reduces the amount of memory required, but increases the time required to look up the next state [Aho et al. 1986].

Lexical analyzers, whether automatically generated or handwritten, must manage their input efficiently. Of course, input is buffered, so that a large batch of characters is obtained at once; then the lexer can process one character at a time in the buffer. The lexer must check, for each character, whether the end of the buffer is reached. By putting a *sentinel* – a character that cannot be part of any token – at the end of the buffer, it is possible for the lexer to check for end-of-buffer only once per token, instead of once per character [Aho et al. 1986]. Gray [1988] uses a scheme that requires only one check per line, rather than one per token, but cannot cope with tokens that contain end-of-line characters. Bumbulis and Cowan [1993] check only once around each cycle in the DFA; this reduces the number of checks (from once per character) when there are long paths in the DFA.

Automatically generated lexical analyzers are often criticized for being slow. In principle, the operation of a finite automaton is very simple and should be efficient, but interpreting from transition tables adds overhead. Gray [1988] shows that DFAs translated directly into executable code (implementing states as case statements) can run as fast as hand-coded lexers. The Flex "fast lexical analyzer generator" [Paxson 1995] is significantly faster than Lex.

EXERCISES

2.1 Write regular expressions for each of the following.

a. Strings over the alphabet $\{a, b, c\}$ where the first a precedes the first b.

b. Strings over the alphabet $\{a, b, c\}$ with an even number of a's.

c. Binary numbers that are multiples of four.

d. Binary numbers that are greater than 101001.

e. Strings over the alphabet $\{a, b, c\}$ that don't contain the contiguous substring baa.

f. The language of nonnegative integer constants in C, where numbers beginning with 0 are *octal* constants and other numbers are *decimal* constants.

g. Binary numbers n such that there exists an integer solution of $a^n + b^n = c^n$.

2.2 For each of the following, explain why you're not surprised that there is no regular expression defining it.

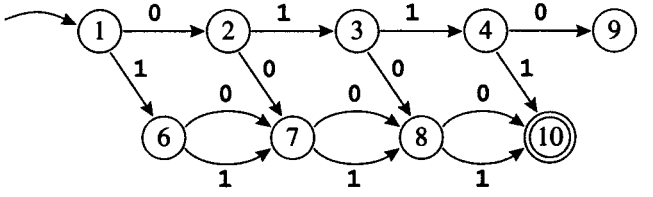
a. Strings of a's and b's where there are more a's than b's.

b. Strings of a's and b's that are palindromes (the same forward as backward).

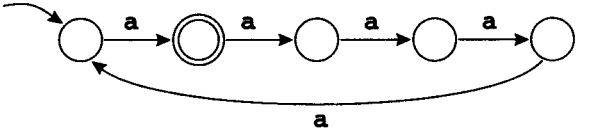
c. Syntactically correct ML programs.

2.3 Explain in informal English what each of these finite state automata recognizes.

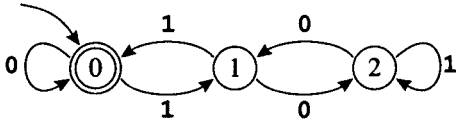
a.



b.



*c.



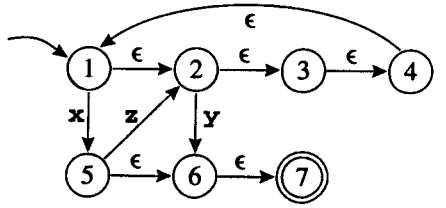
2.4 Convert these regular expressions to nondeterministic finite automata.

a. **(if|then|else)**

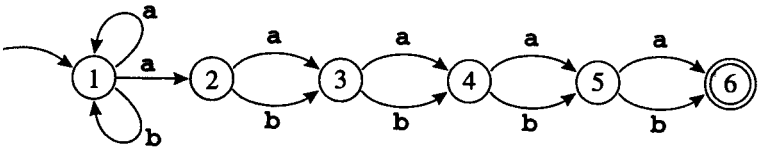
b. **a((b|a*c)x)*|x*a**

2.5 Convert these NFAs to deterministic finite automata.

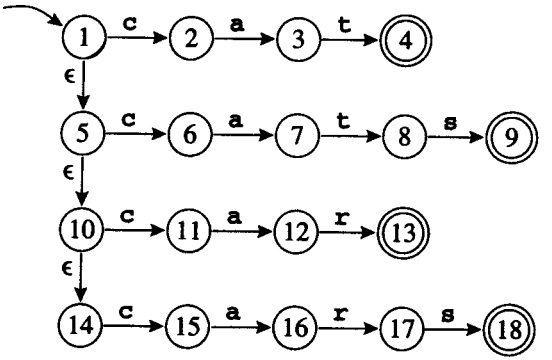
a.



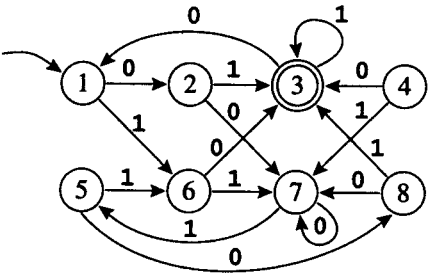
b.



c.



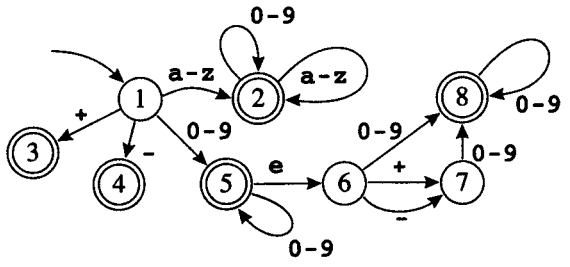
2.6 Find two equivalent states in the following automaton, and merge them to produce a smaller automaton that recognizes the same language. Repeat until there are no longer equivalent states.



Actually, the general algorithm for minimizing finite automata works in reverse. First, find all pairs of inequivalent states. States X, Y are inequivalent if X is final and Y is not or (by iteration) if $X \xrightarrow{a} X'$ and $Y \xrightarrow{a} Y'$ and X', Y' are inequivalent. After this iteration ceases to find new pairs of inequivalent states, then X, Y are equivalent if they are not inequivalent. See Hopcroft and Ullman [1979], Theorem 3.10.

***2.7** Any DFA that accepts at least one string can be converted to a regular expression. Convert the DFA of Exercise 2.3c to a regular expression. **Hint:** First, pretend state 1 is the start state. Then write a regular expression for excursions to state 2 and back, and a similar one for excursions to state 0 and back. Or look in Hopcroft and Ullman [1979], Theorem 2.4, for the algorithm.

***2.8** Suppose this DFA were used by Lex to find tokens in an input file.



a. How many characters past the end of a token might Lex have to examine before matching the token?

b. Given your answer *k* to part (a), show an input file containing at least two tokens such that *the first call* to Lex will examine *k* characters *past the end of the first token* before returning the first token. If the answer to part (a) is zero, then show an input file containing at least two tokens, and indicate the endpoint of each token.

2.9 An interpreted DFA-based lexical analyzer uses two tables,

`edges` indexed by state and input symbol, yielding a state number, and
`final` indexed by state, returning 0 or an action-number.

Starting with this lexical specification,

```
(aba)+      (action 1);
(a(b*)a)    (action 2);
(a|b)       (action 3);
```

generate the `edges` and `final` tables for a lexical analyzer.

Then show each step of the lexer on the string `abaabbaba`. Be sure to show the values of the important internal variables of the recognizer. There will be repeated calls to the lexer to get successive tokens.

****2.10** Lex has a *lookahead* operator `/` so that the regular expression `abc/def` matches `abc` only when followed by `def` (but `def` is not part of the matched string, and will be part of the next token(s)). Aho et al. [1986] describe, and Lex

[Lesk 1975] uses, an incorrect algorithm for implementing lookahead (it fails on `(a|ab)/ba` with input `aba`, matching `ab` where it should match `a`). Flex [Paxson 1995] uses a better mechanism that works correctly for `(a|ab)/ba` but fails (with a warning message) on `zx*/xy*`.

Design a better lookahead mechanism.

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York NY 10011–4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 1998
Revised and expanded edition of *Modern Compiler Implementation in ML: Basic Techniques*
Reprinted with corrections, 1999
First paperback edition 2004

Typeset in Times, Courier, and Optima

A catalogue record for this book is available from the British Library

Library of Congress Cataloguing-in-Publication data

Appel, Andrew W., 1960–
Modern compiler implementation in ML / Andrew W. Appel. – Rev.
and expanded ed.
x, 538 p. : ill. ; 24 cm.
Includes bibliographical references (p. 522–530) and index.
ISBN 0 521 58274 1 (hardback)
1. ML (Computer program language) 2. Compilers (Computer programs)
I. Title.
QA76.73.M6A65 1998
005.4'53—dc21 97-031091
CIP

ISBN 0 521 58274 1 hardback
ISBN 0 521 60764 7 paperback