UNIT-1
Lexical Analysis

The Role of Lexical Analyzer

In the first phase of a compiler, the Lexical Analysis reads the input characters of the source program and produces a sequence of tokens as output. These tokens are sent to the parser for syntax analysis. Whenever the lexical analyzer finds an identifier, it needs to enter that into the symbol table. It will also use the symbol table for knowing the kind of identifier, which helps in determining the proper token. These interactions are shown in Fig. 1. In this, the parser calls the lexical analyzer by *getNextToken* command, then the lexical analyzer reads the characters from its input until it identifies the next token. This token is returned to the parser.

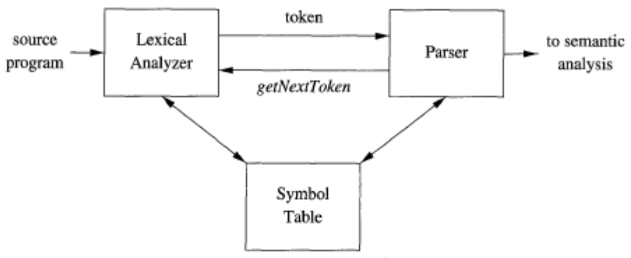


Figure 1: Interactions between the lexical analyzer and the parser

The Lexical analyzer will also perform other tasks like stripping out comments and whitespace, correlating the error messages generated by the compiler with the source program. Sometimes, lexical analyzers are divided into a cascade of two processes:

a) Scanning consists of simple processes, such as deletion of comments and reducing consecutive whitespace characters into one.
b) Lexical analysis, which produces tokens from the output of the scanner.

Lexical Analysis Versus Parsing

The analysis portion of the compiler is separated into lexical and syntax analysis due to a number of reasons.

1. The simplicity of design. If we are designing a new language, separating lexical and syntactic analysis can lead to a clean language design.
2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. Compiler portability is enhanced.

Tokens, Patterns, and Lexemes

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit.

A pattern describes a form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Attributes for Tokens

When more than one lexeme matches a pattern, the lexical analyzer must provide additional information about the particular lexeme for the subsequent phases of a compiler. For example, the pattern for the token *number* matches both 0 and 1, but it is important for the code generator to know which lexeme was found in the source program. Thus, in many cases, the lexical analyzer returns both the token name and attribute value to the parser. The token name can be used for parsing decisions, while the attribute value can be used for the translation of tokens after the parse.

Lexical Errors

Without the help of other components, it is hard for a lexical analyzer to tell, that there is a source-code error. For instance, if the string *fi* is encountered for the first time in a C program in the context:

fi (a = = f(x))

A lexical analyzer cannot tell whether *fi* is a misspelling of the keyword if or an undeclared identifier for a function. Since *fi* is a valid lexeme for the token *id*, the lexical analyzer must return the token *id* to the parser.

However, in some situations, the lexical analyzer is unable to proceed because the prefix of the remaining input will not match with any of the patterns for tokens. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input until we find a token. Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character with another character.
4. Transpose two adjacent characters.

Input Buffering

A two-buffer scheme is used to handle large lookaheads safely. We also use "sentinels" that will save time for checking the ends of buffers.

Buffer Pairs

Specialized buffering techniques will reduce the time required to process a single input character. In this scheme, we use two buffers that are alternately reloaded, as shown in Fig. 2.

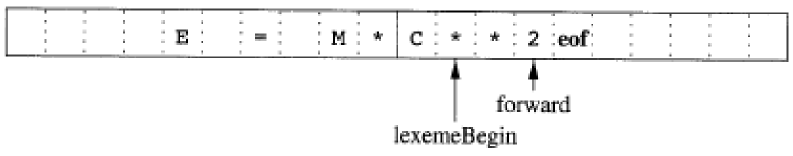


Figure 2: Using a pair of input buffers

Each buffer is of the same size *N*, and *N* is usually the size of a disk block. One read command can load *N* characters into a buffer. If fewer than *N* characters remain in the input file, then a special character called *eof*, marks the end of the source file.

Two pointers to the input are maintained:

1. Pointer *lexemeBegin*, marks the beginning of the current lexeme
2. Pointer *forward* scans ahead until a pattern match is found

Once the next lexeme is determined, *forward* is set to the right end of the character. Then we transform the lexeme to a token, and it is returned to the parser. The *lexemeBegin* is set to the character immediately after the lexeme just found. In Fig. 2, we see forward has passed the end of the next lexeme, ** (exponentiation operator), and must be retracted one position to its left.

Advancing the *forward* is required to first test whether we have reached the end of one of the buffers. If we have reached the end of the buffer, we reload the input to the other buffer and move *forward* to the beginning of the newly loaded buffer. We will not overwrite the lexeme in its buffer until we determine it.

Sentinels

In this scheme, for each character read we make two tests: one for the end of the buffer, and one to determine what is the character. We can combine both the tests by placing a *sentinel* character at the end of each buffer. The sentinel is a special character that cannot be part of the source program that is *eof*.

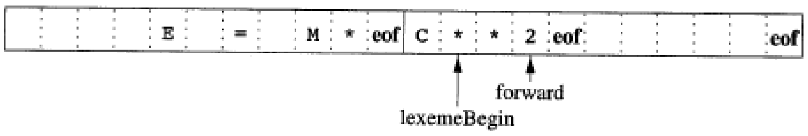
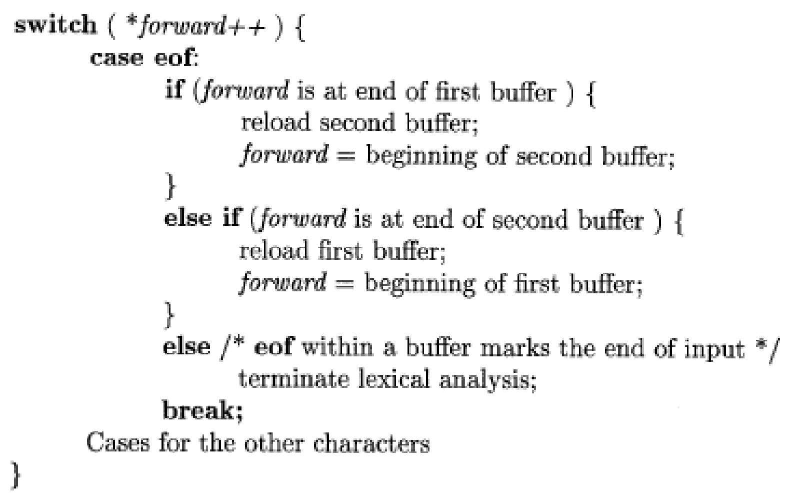


Figure 3 : Sentinels at the end of each buffer

Figure 3 shows the Sentinels at the end of each buffer. The *eof* is also used as a marker for the end of the entire input. Any *eof* that appears other than at the end of a buffer means that the input is at an end. The algorithm for advancing forward is given in Fig. 4.



```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Figure 4: Lookahead code with sentinels

Specification of Tokens

The important notation for specifying patterns of a lexeme are regular expressions. While they cannot express all patterns, but they are very effective in specifying those types of patterns.

Strings and Languages

An alphabet is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0, 1} is the binary alphabet.

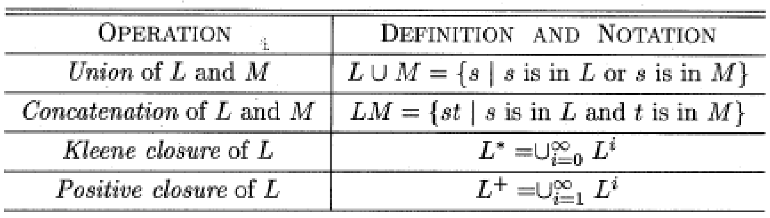
A string over an alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string s is written as |s|. The empty string, denoted ϵ, is the string of length zero.

A language is any countable set of strings over some fixed alphabet. Abstract languages like ∅, the empty set, or {ϵ}, the set containing only the empty string, are languages under this definition.

If *x* and *y* are strings, then the concatenation of *x* and *y*, denoted *xy*, is the string formed by appending *y* to *x*. If we think of concatenation as a product, we can define the "exponentiation" of strings as follows. Define s^0 to be ϵ, and for all i > 0, define s^i to be $s^{i-1}s$. Since ϵs = s, it follows that s^1 = s. Then s^2 = ss, s^3 = sss, and so on.

Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 5



OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 5: Definitions of operations on languages

Regular Expressions

Identifiers are described by using sets of letters and digits and using the language operator's union, concatenation, and closure. This process is useful for regular expressions to describe all the languages that can be built from these operators. In this notation, if *letter_* is established to stand for any letter or the underscore, and *digit* is established to stand for any digit. We could describe the identifiers in C language by: *letter_ (letter_* | *digit)* *

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression *r* denotes a language *L(r)*. Here are the rules that define the regular expressions over some alphabet ∑ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ϵ is a regular expression, and *L(ϵ)* is {ϵ}, that is, the language is the empty string.
2. If *a* is a symbol in ∑, then *a* is a regular expression, and *L(a)* = {*a*}, that is, the language with one string *a*.

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose *r* and *s* are regular expressions denoting languages *L(r)* and *L(s)*, respectively

1. *(r)* | *(s)* is a regular expression denoting the language *L(r)* U *L(s)*.
2. *(r)(s)* is a regular expression denoting the language *L(r)L(s)*.

3. *(r)** is a regular expression denoting *(L (r))* *.
4. *(r)* is a regular expression denoting *L(r)*. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has the highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has the lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression (a) | ((b)*(c)) by a | b*c. Both expressions denote the set of strings.

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write *r* = *s*. For instance, (a | b) = (b | a).

Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\dots \\ d_n &\rightarrow r_n \end{aligned}$$

where:

1. Each d_i is a new symbol, not in ∑ and distinct from other *d*'s, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

In C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

*letter_* → A | B | · · · | Z | a | b | · · · | z | _

digit → 0 | 1 | · · · | 9

id → letter_ (letter_ | digit) *

Extensions of Regular Expressions

Many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions.

1. One or more instances. The unary, postfix operator + represents the positive closure of a regular expression and its language.
2. Zero or one instance. The unary postfix operator ? means "zero or one occurrence." That is, r? is equivalent to r | ϵ. The ? operator has the same precedence and associativity as * and +.
3. Character classes. A regular expression $a_1 \mid a_2 \mid \cdots \mid a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2 \cdots a_n]$. More importantly, when $a_1, a_2, \cdots, a_n$ form a logical sequence, we can replace them by a_1- a_n

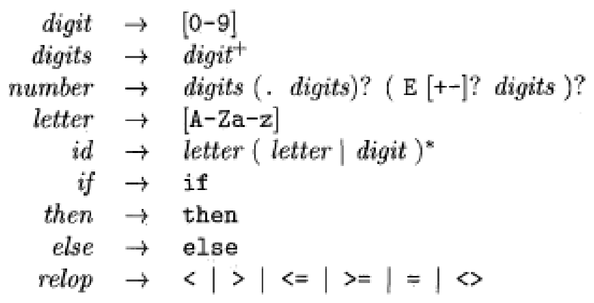
Recognition of Tokens

Take patterns for all the needed tokens and build a piece of code that examines the input string and finds a lexeme matching one of the patterns. We make use of the following running example.

$$
\begin{array}{lcl}
\mathit{stmt} & \to & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
 & \mid & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\
 & \mid & \epsilon \\
\mathit{expr} & \to & \mathit{term} \textbf{ relop } \mathit{term} \\
 & \mid & \mathit{term} \\
\mathit{term} & \to & \textbf{id} \\
 & \mid & \textbf{number}
\end{array}
$$

Figure 6: A grammar for branching statements

The terminals of the grammar are *if*, *then*, *else*, *relop*, *id*, and *number*, are the names of the token. The patterns for these tokens are described using regular definitions, as shown in Fig. 7.



digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

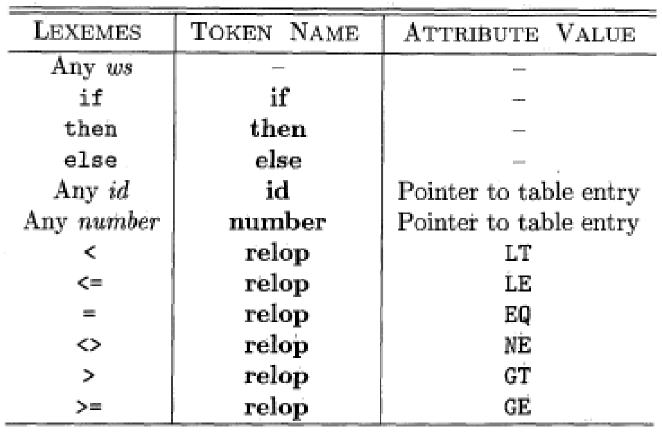
Figure 7: Patterns for tokens

We make an assumption that keywords are reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In lexical analyzer the job of stripping out whitespace was defined by:

ws ➔ (*blank* | *tab* | *newline*) +

Token *ws* is different from the other tokens, and we do not return it to the parser. Our goal for the lexical analyzer is summarized in Fig. 8. That table shows, for each lexeme, which token name and attribute value are returned to the parser.



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Figure 8: Tokens, their patterns, and attribute values

Transition Diagrams

As an intermediate step in the construction of a lexical analyzer, we first convert patterns into flowcharts, called "transition diagrams." In this, we perform the conversion from regular-expression patterns to transition diagrams. *Transition diagrams* have a collection of nodes, called states. Each state represents a condition.

Edges are directed from one state of the transition diagram to another. Each edge is *labeled* by a symbol or set of symbols. If we are in some state *s*, and the next input symbol is *a*, we look for an edge out of state *s* labeled by *a*. If we find such an edge, we advance the *forward* pointer and enter the state of the transition diagram to which that edge leads. We shall assume that all our transition diagrams are *deterministic*. Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found. We always indicate a final state by a double circle, and if there is an action to be taken, we shall attach that action to the final state.
2. In addition, if it is necessary to retract the forward pointer one position, then we shall additionally place a * near that final state.
3. One state is designated the *start state*, or *initial state*; it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Figure 9 is a transition diagram that recognizes the lexemes matching the token *relop*.

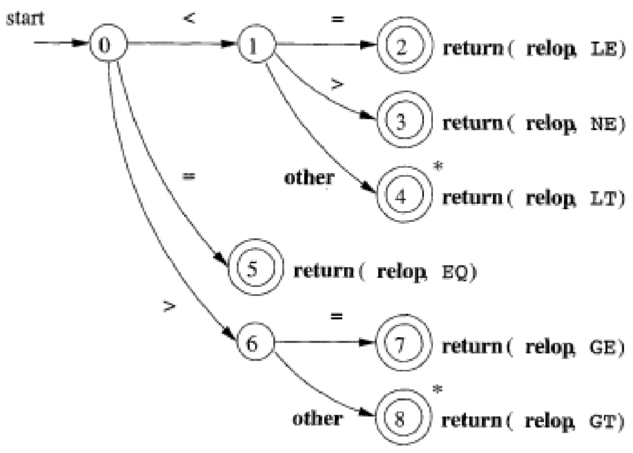


Figure 9: Transition diagrams for *relop*

Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like *if* or *then* are reserved, so they are not identifiers even though they look like identifiers. Thus, although we typically use a transition diagram like that of Fig. 10 to search for identifier lexemes, this diagram will also recognize the keywords *if*, *then*, and *else* of our running example

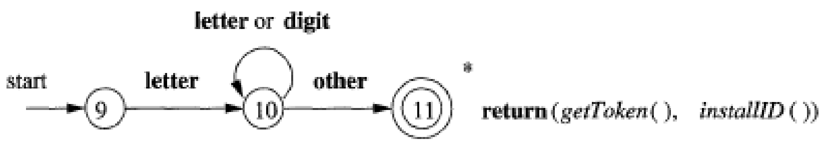


Figure 10: Transition diagram for *id*'s and Keywords

There are two ways that we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in Fig. 10. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. The function getToken examines the symbol table entry for the lexeme found and returns the token name as either id or keyword.
2. Create separate transition diagrams for each keyword; an example for the keyword *then* is shown in Fig. 11. In this, the "nonletter-or-digit" is any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended or not otherwise the wrong token will be returned. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to *id*, when the lexeme matches both patterns. We do not use this approach in our example.

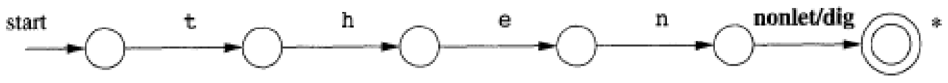


Figure 11: Hypothetical transition diagram for the keyword *then*

Completion of the Running Example

The transition diagram for token number is shown in Fig. 12. If we see an E, then we have an "optional exponent".

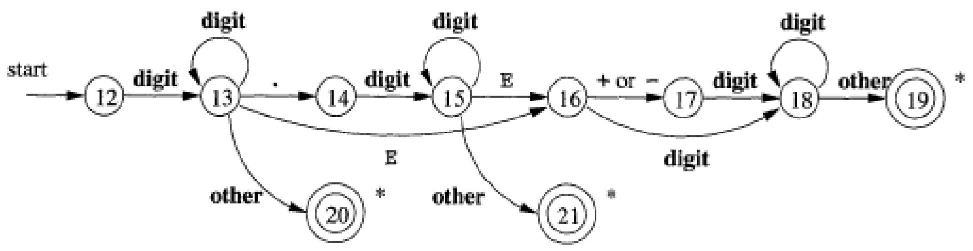


Figure 12: A transition diagram for unsigned numbers

The final transition diagram, shown in Fig. 13, is for whitespace. In that diagram, we look for one or more "whitespace" characters, represented by delim in that diagram - typically these characters would be blank, tab, and newline.

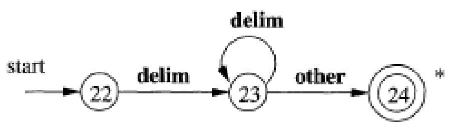


Figure 13: Transition diagram for whitespace

Architecture of a Transition-Diagram-Based Lexical Analyzer

A collection of transition diagrams can be used to build a lexical analyzer. Each state is represented by a piece of code. A variable called *state* is holding the number of the current state for a transition diagram. A switch based on the value of *state* takes us to code for each of the possible states, where we find the action of that state. Often, the code for a state determines the next state by reading and examining the next input character.

In Fig. 14 we see a sketch of getRelop( ), is a C++ function and returns the token name and an attribute value. getRelop( ) first creates a new object retToken and initializes its first component to RELOP. A function nextChar( ) obtains the next character from the input and assigns it to the local variable *c*. If the next input character is not a comparison operator, then a function fail( ) is called. The retract( ) is to retract the input pointer one position.

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 14: Sketch of implementation of relop transition diagram

Let us consider the ways code like Fig. 14 could fit into the entire lexical analyzer.

1. We could arrange the transition diagrams for each token to be tried sequentially. Then, the function *fail*( ) resets the pointer *forward* and starts the next transition diagram , each time it is called. This method allows us to use transition diagrams for the individual keywords. We need to use these before we use the diagram for *id*, in order for the keywords to be reserved words.
2. We could run the various transition diagrams "in parallel". If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern.
3. The preferred approach is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matches any pattern. In general, the problem of combining transition diagrams for several tokens is more complex.

The Lexical - Analyzer Generator Lex (or) A Language for Specifying Lexical Analyzer

Lex is a tool that allows one to specify a lexical analyzer. The input notation for the *Lex* tool is *Lex language* and the tool itself is the *Lex compiler*. The Lex compiler transforms the input patterns into a transition diagram and generates code that is placed in a file called *lex.yy.c*.

Use of Lex

Fig. 15 shows how the *Lex* is used. An input file *lex.l*, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms *lex.l* to a C program and keeps in a file that is always named as *lex.yy.c*. Later the file is compiled by the C compiler into a file called *a.out*. The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

The *a.out* is used as a subroutine of the parser that returns an integer, which is a code for one of the token names. The attribute value of a token is placed in a global variable *yylval*, which is shared between the lexical analyzer and parser. Therefore, it is simple to return both the token name and an attribute value

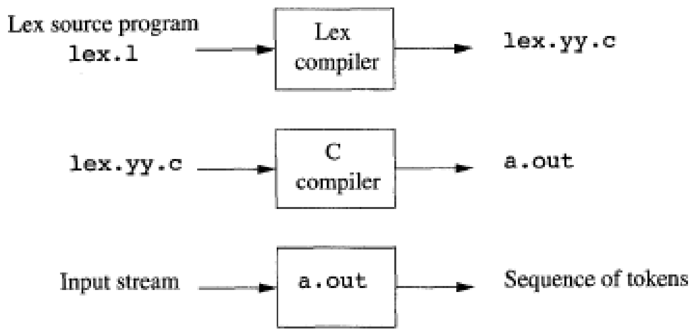


Figure 15: Creating a lexical analyzer with Lex

Structure of Lex Programs

A Lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions

The declarations section includes declarations of *variables*, *manifest constants* (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14) and *regular definitions*.

In *Regular Definitions*, we assign names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If ∑ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\ldots \\ d_n &\rightarrow r_n \end{aligned}$$

where

1. Each d_i is a new symbol and distinct from any other d's, not in ∑ and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

The translation rules each have the form

Pattern { Action }

Each pattern is a regular expression, which may use the regular definitions. The actions are fragments of code. The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by Lex will behave as follows. When the parser calls the lexical analyzer, it will read the input, one character at a time, until it finds the patterns P_i. It then executes the associated action A_i. The A_i will return to the parser. The lexical analyzer will return a token name to the parser. If there is a need to pass the additional information about the lexeme found, then it will make use of the shared integer variable *yylval*.

Conflict Resolution in Lex

When several prefixes of the input match one or more patterns then it will use the following two rules to select the proper token:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

The Lookahead Operator

Lex automatically reads one character ahead of the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input. Sometimes, input can be matched with a certain pattern when it is followed by certain other characters. In this case, we make use of slash to indicate the end part of the matched pattern. What follows / is an additional pattern. The additional pattern must be matched before we decide the token in the question.

For example, in FORTRAN and some other languages, keywords are not reserved. That situation creates problems, such as a statement

IF(I, J) = 3

where IF is the name of an array, not a keyword. This statement doesn't match with the statements of the form

IF (condition) THEN . . .

where IF is a keyword. Fortunately, we can be sure that the keyword IF is always followed by a left parenthesis, the condition, a right parenthesis and a letter. Thus, we could write a Lex rule for the keyword IF like:

```
IF / \( .* \) {letter}
```

This rule says that the pattern matches the lexeme is IF. The characters followed by slash is the additional pattern. In this pattern, the first character is the left parentheses. Since that character is a Lex metasymbol, it must be preceded by a backslash to indicate that it has its special meaning. The dot and star match "any string without a newline." It is followed by a right parenthesis, again it must be preceded by a backslash to indicate that it has its special meaning. The additional pattern is followed by the symbol *letter*.