Chapter 2:

LEXICAL ANALYSIS

2. 1 The role of the Lexical Analyser:

As the first phase of a compiler, is to read the input characters of the source program group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.

The stream of tokens is sent to the parse for syntax analysis. It is common for the lexical analyser to interact with the symbol table as well. When the lexical the lexical analyser discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.

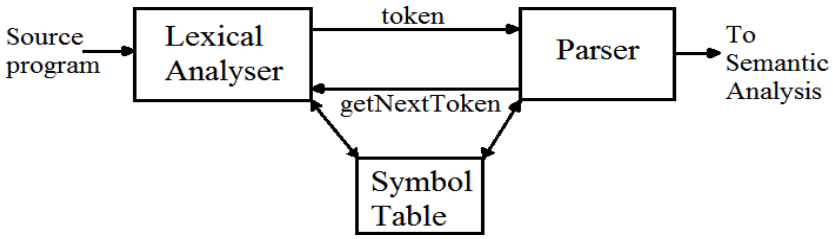


Fig: Interaction between the lexical analyser and parser.

The interaction between the lexical analyser and the parser are shown in the above figure.

Commonly the interaction is implemented by having the parser call the lexical analyser. The call, suggested by the „getNextToken" command causes the lexical analyser to read characters from its input until it can identify the next lexeme and produce for it new token, when it returns to the parser.

The lexical analyser performs certain other task besides identification of lexemes.

1. One such task is stripping out comments and white space.
2. Correlating error messages generated by the compiler with the source program.

Sometimes, lexical analysers are divided into a cascade of two processes:

1. Scanning consists of the simple process that does not require tokenization of the input, deletion of comments and compaction of consecutive whitespace characters into one.
2. Lexical analysis proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Lexical Analysis versus Parsing:

There are number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing phases.

1. Simplicity of design is the most important consideration:
   The separation of lexical and syntactic analysis often allows us to simplify at least one of their tasks.
2. Compiler efficiency is improved:
   A separate lexical analyser allows us to apply specialised technique that serves only the lexical task, not the job of parsing.
3. Compiler portability is enhanced:
   Input device specific peculiarities can be restricted to the lexical analyser.

Tokens, Patterns and Lexemes

Tokens: A token is a pair consisting of a token name and an optional attribute values.
Pattern: A pattern is a description of the form that the lexeme of a token may take.
Lexeme: A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token.

2.2 Input Buffering:

2.2.1 Buffer Pairs:

Because of the amount of the time taken to process characters and the largest number of characters must be processed during the compilation of a large source program, specialized buffering technique have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded, as suggested in below figure.

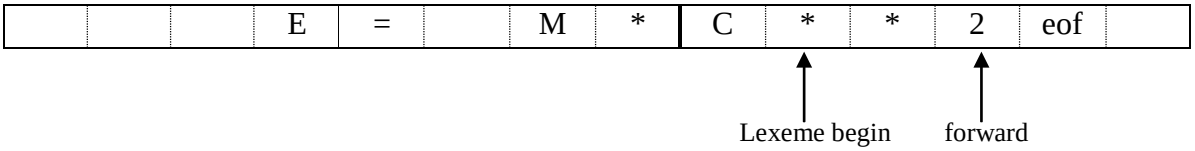


Fig: Using a pair of input buffers

Each buffer is of same size N and N is usually the size of the disk block E.g.: 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character represented by „*eof*" marks the end of the source file.

Two pointers to the input are maintained.

1. Pointer *lexeme Begin* marks the beginning of the current lexeme.
2. Pointer *forward* scans ahead until a pattern match is formed.

Once the next lexeme is determined, forward is set to the character at its right end. Lexeme Begin is set to the character immediately after the lexeme just found.

2.2.2 Sentinels:

Each time we advance forward, that we have not moved off one of the buffers, if we do, then we must also reload the other buffer.

Thus for each character read, we make two tests

1. One for the end of the buffer and
2. One to determine what character is read.

We can combine the buffer end test with the test for the current character if we explain each buffer to hold a sentinel character at the end. The sentinel is the special character that cannot be the part of source programmer and a natural choice is the character eof .

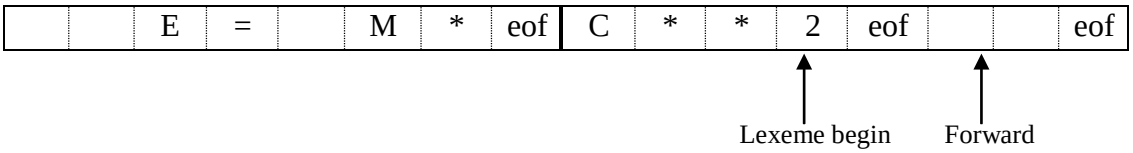


Fig: Sentinels at the end of each buffer

```
Switch (*forward++)
{
case of :
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
terminate lexical analysis;
break;
case for the other characters.
}
```

Fig: look ahead code with sentinels

2.3 Specification of Token:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying these types of patterns that we actually need for tokens.

2.3.1 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. ASCII is an important example of an alphabet.

A string over an alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string S, usually return mod(s), is the number of occurrence of symbols in S.

For example: Banana is a string of length 6. The empty string, denoted ϵ, is the string of length 0.

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

2.3.2 Operations on languages

The most important operations on languages are union, concatenation and closure.

- The union of language is a collection of strings, is the string of all distinct elements in the collection.
- The Concatenation of languages is all strings formed by taking a string from the first language.
- The kleene closure of a language L, denoted L* is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of zero times".
- The positive closure, denoted by L^+ is the set of strings you get by concatenating L one or more times.

OPERATION	DEFINATION & NOTATION
Union of L and M	L U M = {S| S is in L or S is in M}
Concatenation of L and M	LM = {ST | S is in L and T is in M}
Kleene closure of L	$L^* = U_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = U_{i=0}^{\infty} L^i$

2.3.3 Regular expressions

Regular expression is a Meta language, use to describe the particular pattern of interest.

Ex: - letter_ (letter_|digit)*

In the above example letter_ is established to stand for any letter or the underscore and digit_ is established to stand for any digit. The vertical bar above means union, the parenthesis are used to group sub expressions, the star means "Zero or more occurrence of ".

The rules for the regular expressions are:

BASIS: - There are two rules from the basis.

1. ε is a regular expression, and L(ε) is {ε},that is, the language where sole member 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 of length one with „a" in its one position.

INDUCTIONS:-There are four parts to the induction where by large regular expressions are built from smaller ones.

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).

LAW	DESCRIPTION
r|s = s|r	| is commutative
r| (s|t) = (r|s) |t	| is associative
r (st) = (rs) t	concatenation is associative
r(s|t) = rs |rt ; (s|t)r = sr|tr	concatenation distributers over |
εr = rε = r	ε is the identity for concatenation.
r*= (r|ε)*	ε is the guaranteed in a closure.
r** = r*	* is idempotent.

Fig: - algebraic laws for regular expression.

2.4 Transition Diagrams:

As an intermediate step in the construction of a lexical analyzer first convert patterns into stylized flowcharts, called “transition diagrams."

Transition diagrams have a collection of nodes or **circles,** called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

Transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Some important conventions about transition diagrams are:

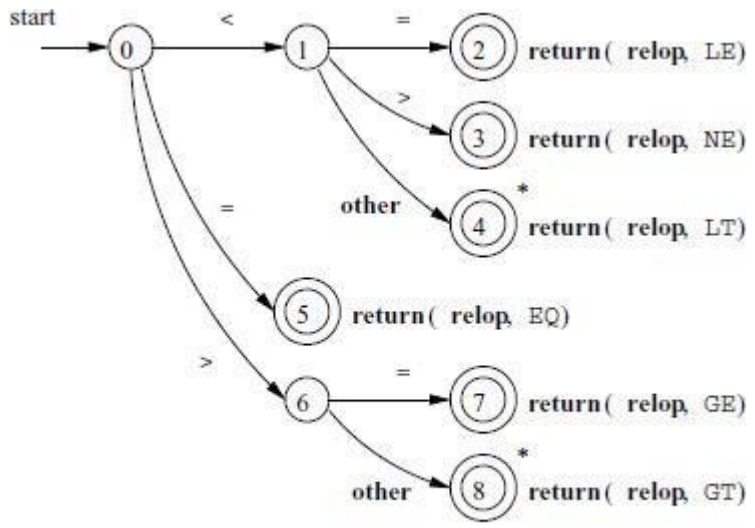
1. Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken | typically returning a token and an attribute value to the parser | we shall attach that action to the accepting state.

2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting 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.

Example:

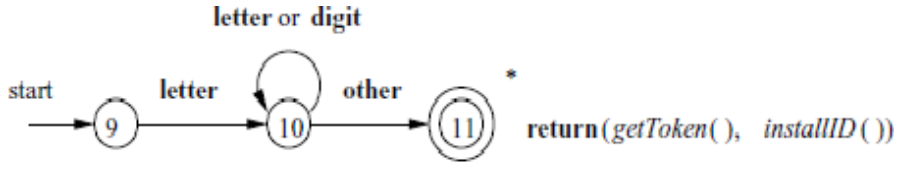
Transition diagram that recognizes the lexemes matching the token **relop**. We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, < >, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme < >, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.



Transition diagram for **relop**

Recognition of Reserved Words and Identifiers:

Usually keywords like if or then are reserved so they are not Identifiers even though they look like identifiers.

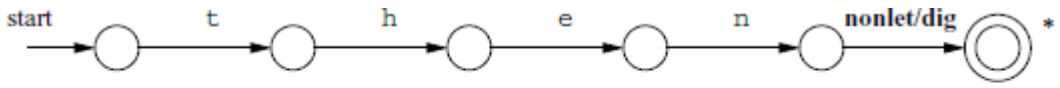


A transition diagram for **id**'s and keywords

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

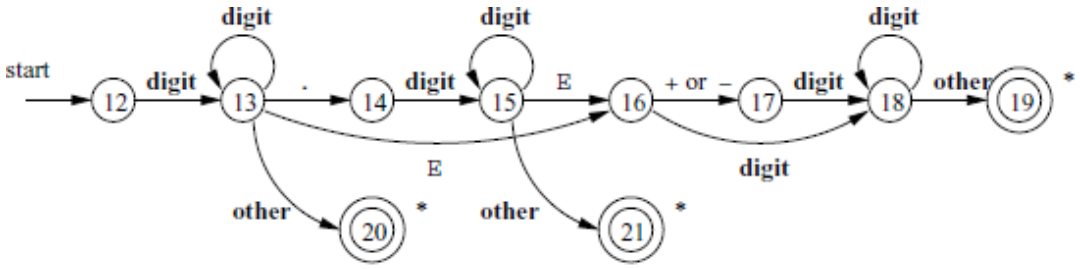
1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. We have supposed that this method is in use in above Figure. When we find an identifier, a call to installID 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.

2. Create separate transition diagrams for each keyword; an example for the keyword then is shown in Figure. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier.



Hypothetical transition diagram for the keyword then

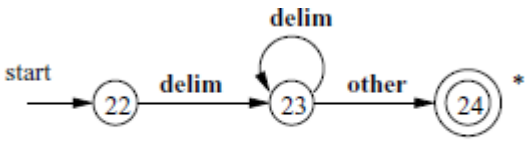
Transition diagram for unsigned numbers:



A transition diagram for unsigned numbers

Transition diagram 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, newline, and perhaps other characters that are not considered by the language design to be part of any token.



A transition diagram for whitespace