MODULE - I
INTRODUCTION TO COMPILERS AND LEXICAL ANALYSIS

SYLLABUS:

Analysis of the source program - Analysis and synthesis phases, Phases of a compiler. Compiler writing tools. Bootstrapping. Lexical Analysis - Role of Lexical Analyser, Input Buffering, Specification of Tokens, Recognition of Tokens.

1. INTRODUCTION

All softwares are written in some programming languages. Before this program can be run, it must be translated into a form in which it can be executed by computers. This translation is done by a software system known as *Compilers.*

Language Processors

- A compiler is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language)
- An important role of the compiler is to report any errors in the source program that it detects during the translation process

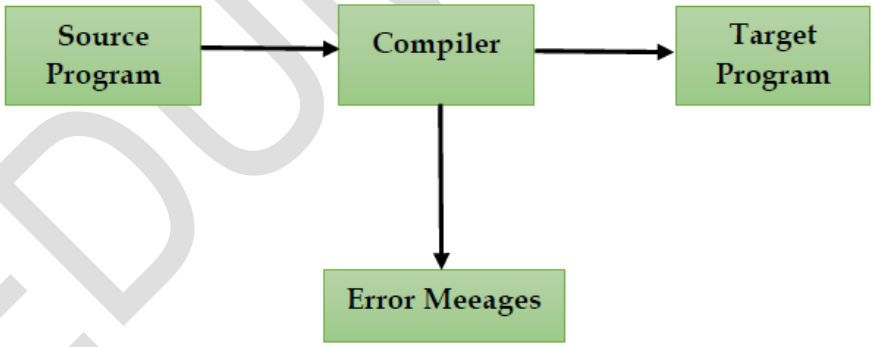


Fig 1.1: A Compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs. (Fig. 1.2)

Fig 1.2: Running the target program

An interpreter is another kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

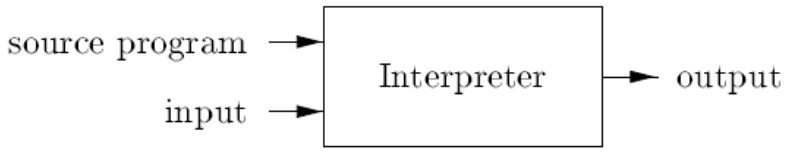


Fig 1.3: An Interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

2. ANALYSIS OF SOURCE PROGRAM

The below figure shows a Language Processing System (fig 2.1). There we can see that, in addition to compilers, we need several other programs to create an executable target program.

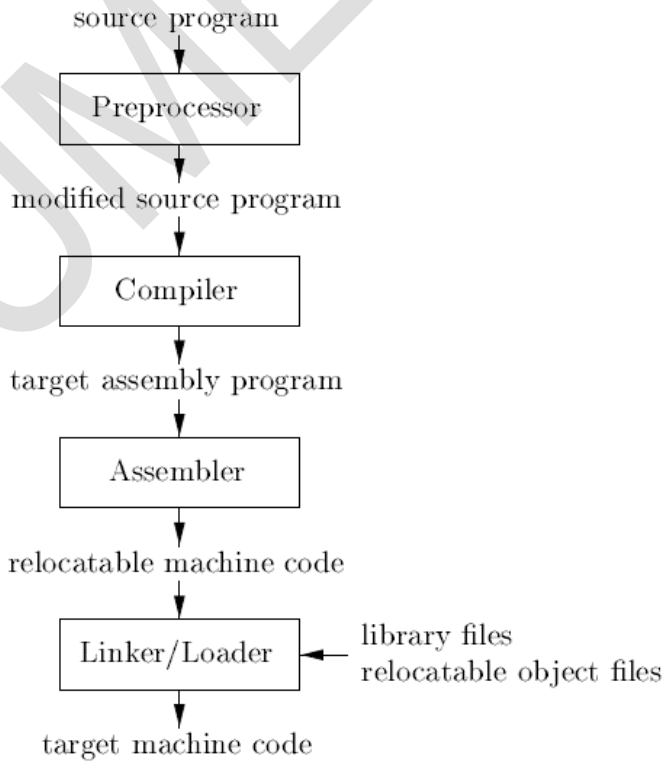


Fig 2.1: A language processing system

The steps involved in the analysis of source program are:

- **Step 1: Preprocessor**
  Source program acts as the input to the preprocessor. A source program may be divided into modules stored in separate files. The pre-processor modifies the source program by replacing the header files with the suitable content. This is known as file inclusion. It also performs macro-processing, that is, expansion of macros into source language statements. This modified source program is then fed to the compiler.

- **Step 2: Compiler**
  The compiler translates the modified source program in high level language into the target program. If the target program is in machine language, then it can be executed directly. If the target program is in assembly language, then it is fed to the assembler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

- **Step 3: Assembler**
  The assembler translates the assembly language code into the relocatable machine code.

- **Step 4: Linker/Loader**
  Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. This task is performed by the Linker. The Loader loads this integrated code into memory for execution. The output of the Linker/Loader is the equivalent machine language code of the source code.

3. PHASES OF A COMPILER (or STRUCTURE OF A COMPILER)

A compiler contains two parts for the conversion of source program into the target code:

- Analysis part (*front end*)
- Synthesis part (*back end*)

Analysis part:

- Breaks the source program into constituent pieces and imposes a grammatical structure on them
- This structure is then used to create an intermediate representation of the source program

- If any errors (syntactic or semantic) are found during this transformation, the analysis part provide informative messages to the user about the same
- It also collects information about the source program, and stores it in a data structure called ***Symbol Table***, which is passed along with the intermediate representation to the synthesis part

Synthesis part:

- Synthesis part constructs the target code from the intermediate code and information from the symbol table

The compilation process is a sequence of ***phases***, each of which transforms one representation of source program into another. A typical decomposition of compiler into different phases is shown in the below figure. The symbol table is used by all the phases of the compiler. Note that the lexical analysis, syntax analysis, semantic analysis and the intermediate code generation phase constitutes the Analysis part and the remaining phases builds up the Synthesis part.

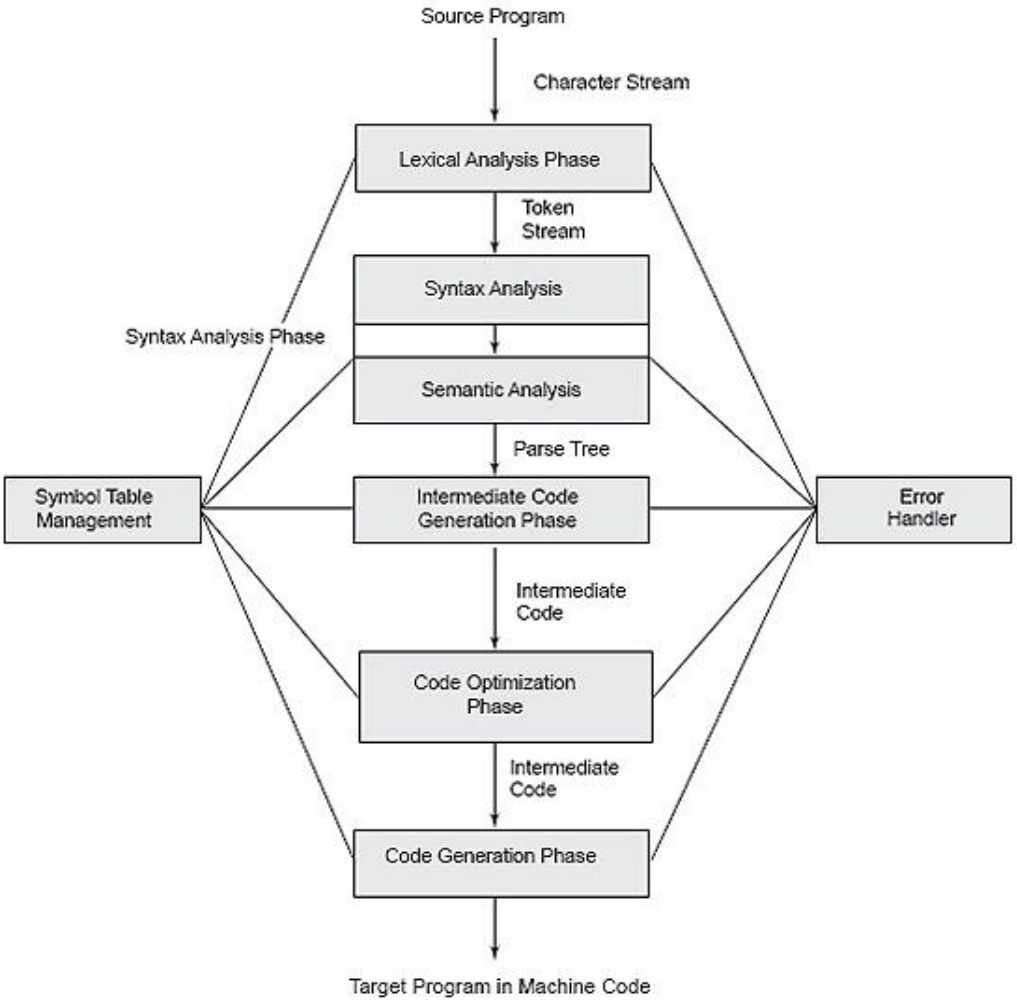


Fig 3.1: Phases of a compiler

Lexical Analysis: (or Scanning or tokenizing)

- The first phase of a compiler
- This phase scans the source program as a stream of characters and converts it into meaningful *lexemes*
- For each lexeme, the lexical analyser produces as output a *token* of the form

< *token_name , attribute_value* >

- This token is then passed to the syntax analysis phase.
- In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens:

1. **position** is a lexeme that would be mapped into a token <id, 1>, where **id** is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol- table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol **=** is a lexeme that is mapped into the token < = >. Since this token needs no attribute-value, we have omitted the second component.
3. **initial** is a lexeme that is mapped into the token < id, 2> , where 2 points to the symbol-table entry for initial .
4. **+** is a lexeme that is mapped into the token <+>.
5. **rate** is a lexeme that is mapped into the token < id, 3 >, where 3 points to the symbol-table entry for rate.
6. ***** is a lexeme that is mapped into the token <* > .
7. **60** is a lexeme that is mapped into the token <60>

Blanks separating the lexemes would be discarded by the lexical analyzer.
Thus the representation of the assignment statement,

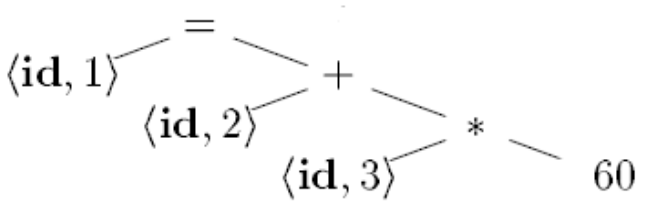
position = initial + rate * 60

after lexical analysis as the sequence of tokens as:

< id, l > < = > <id, 2> <+> <id, 3> < * > <60>

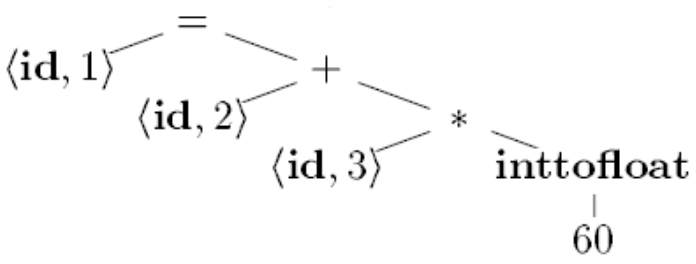
Syntax Analysis: (or Parsing)

- The second phase of compiler
- Uses the tokens created by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream
- A typical representation is a syntax tree (also known as a parse tree) in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- The syntax tree for above token stream is:



Semantic Analysis:

- It checks whether the parse tree constructed in the syntax analysis phase follows the rules of the language.
- An important part of semantic analysis is ***type checking***, where the compiler checks that each operator has matching operands.
- The language specification may permit some type conversions called ***coercions.*** For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating point numbers. If the operator is applied to a floating point number and an integer, the compiler may convert or coerce the integer into a floating point number.
- Suppose that in our example *position, initial* and *rate* are declared as floating point numbers. Then the type checker will discover that the operator * is applied to a floating point number *rate* and an integer 60. In that case, the integer 60 will be converted to floating point number using ***inttofloat***
- Finally, the semantic analyzer will produce an annotated syntax tree as its output



Intermediate Code Generation:

While translating the source program into the target code, a compiler may construct one or more intermediate representations. Syntax trees are a form of intermediate representation, produced during the syntax analysis and the semantic analysis.
After syntax and semantic analysis, many compilers generate an explicit low-level or machine-like intermediate representation. This intermediate representation has two properties:

- It should be easy to produce
- It should be easy to translate into the target machine code

Here we consider an intermediate representation called ***three-address code,*** which consists of a sequence of assembly-like instructions with three operands per instruction. The output three-address code for our previous example is,

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

In three-address code,

1. Each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done. For e.g. multiplication precedes addition in our source program.
2. The compiler must generate a temporary name to hold the value computed by a three-address instruction
3. Some three-address instructions (first and last instructions in our example), have fewer than three operands

Code Optimization

The machine independent code optimization phase attempts to improve the intermediate code so that better target code will result (faster, shorter code or target code that consumes less power). In our previous example, the code can be optimized as given below.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

There is a great variation in the amount of optimization done in different compilers. "Optimizing Compilers" are those compilers that do the most optimization.

Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then the intermediate instructions are translated into sequence of machine instructions that perform the same task. The optimized intermediate code in our example can be converted to the following machine code.

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

The first operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating-point numbers.

Symbol Table Management

The symbol table is a data structure containing a record for each variable name used in the source program, along with its attributes. These attributes may provide information about the storage allocated for a name, its type, its scope, and in case of procedure names, things such as the number and types of its arguments, the method of passing each argument, and the return type. The symbol table should be designed to allow the compiler to find the record for each name quickly, and to store and retrieve data from that record quickly.
Symbol table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer. In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer returns a token, say ***id***, to the parser. Only the parser can decide whether to use a previously created symbol-table entry or create a new one for this identifier.

Error Handling

Each phase of compiler may encounter errors. However, after detecting an error, the phase must somehow deal with that error so that the compilation can proceed, allowing further errors in the source program to be detected. The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error. During this whole process processing time of the program should not be slow.

Functions of Error Handler:

- Error Detection
- Error Report
- Error Recovery

```
Error handler=Error Detection+Error Report+Error Recovery
```

Errors can be of two types:

- Compile-time errors
- Run-time errors

Classification of Compile-time errors

1. **Lexical** : This includes misspellings of identifiers, keywords or operators
2. **Syntactical** : a missing semicolon or unbalanced parenthesis
3. **Semantical** : incompatible value assignment or type mismatches between operator and operand
4. **Logical** : code not reachable, infinite loop.

4. COMPILER WRITING (CONSTRUCTION) TOOLS

Compiler writers use software development tools and more specialized tools for implementing various phases of a compiler. Some commonly used compiler construction tools include the following.

- **Scanner generators:** These generators produce lexical analyzers from a regular expression description of the tokens of a language.
- **Parser generators:** They automatically produce syntax analyzers from a grammatical description of a programming language.
- **Syntax-directed translation engines:** Produce collections of routines for walking a parse tree and generating intermediate code.
- **Automatic code generators:** This generator takes an intermediate code as input, and converts each operation of the intermediate code into the equivalent machine language.
- **Data-flow analysis Engines:** It facilitates the gathering of information about how values are transmitted from one part of the program to other part. It is a key part of code optimization.
- **Compiler construction toolkits:** Provides an integrated set of routines for constructing various phases of a compiler.

5. BOOTSTRAPPING

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.

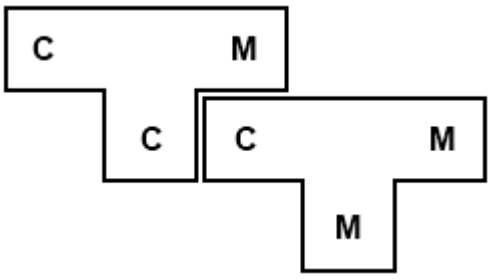
Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.

A compiler is characterized by 3 languages:

1. Its source language
2. Its object language (or the target language)
3. The language in which it is written (or the implementation language)

We use T-diagrams (Tombstone diagram) to represent a compiler. (C_L^{ST})

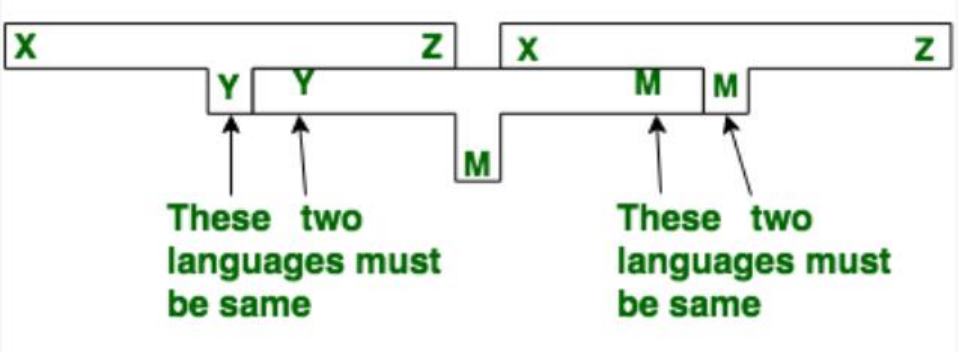
Suppose that we have a compiler implemented in C language, to translate C language to Machine code. But let's consider that our machine can understand only machine code. So we have to design another complier which is implemented in machine language. (Figure given below)



To clearly understand the bootstrapping technique, consider the following scenario.

Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

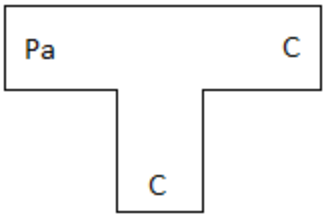
Following diagram illustrates the above scenario.



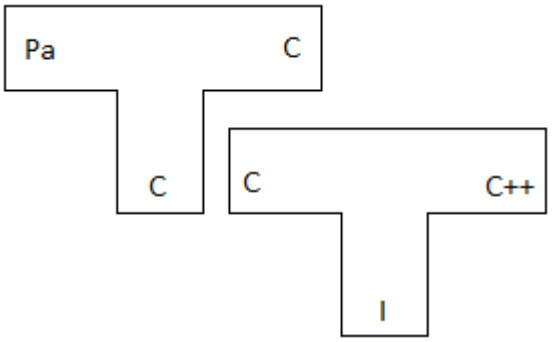
Example:

Suppose we have a Pascal translator written in C language that takes Pascal code as input and produces C code as output. Now create a Pascal translator for the same in C++.

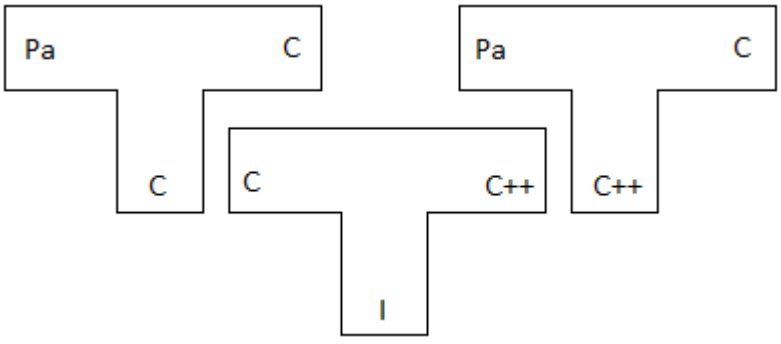
Step 1: Write a compiler for Pascal to C conversion, implemented in C language.



Step 2: Write another compiler to translate C to C++, implemented in any language, say I.



Step 3: Finally, we compile the first compiler using the second compiler



Step 4: Thus we created a compiler written in C++ language, to convert Pascal code to C code.

Cross Compiler: A cross compiler is a compiler that runs on one machine, but produces output for another machine. That is, a compiler capable of creating an executable code for a platform other than the one on which the compiler is running.

6. LEXICAL ANALYSIS

6.1 The Role of Lexical Analyzer

- The main task of lexical analyzer 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 then sent to the parser for syntax analysis.
- The lexical analyzer interacts with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- These interactions are shown in the below figure. These interactions are implemented by having the parser call the lexical analyzer. The call using, *getNextToken* command, causes the lexical analyzer to read characters from the input until it can identify the next lexeme and produce the token for it.

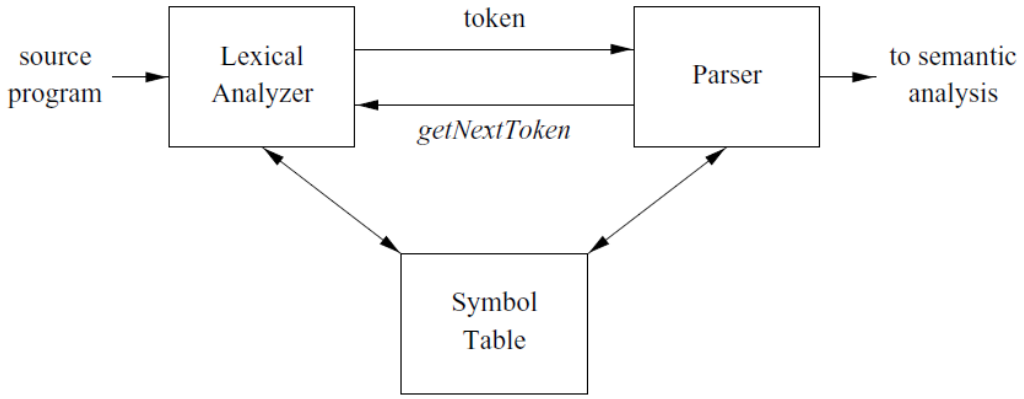


Fig 6.1: Interactions between the lexical analyzer and the parser

Besides the identification of tokens, the other tasks of a lexical analyzer are:

- Stripping out comments and white spaces (blank, newline, tab etc)
- Correlating error messages with the source program. That is, display error messages with its occurrence by specifying the line number.
- Expands macros if it is found in the source program

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

- ***Scanning:*** A simple process that do not require tokenization of input, such as deletion of comments and compaction of consecutive white space characters into one
- ***Lexical Analysis:*** Produces tokens as output

Lexical Analysis vs Parsing

Reasons why the analysis portion of a compiler is normally separated into lexical analysis (scanning) and syntax analysis (parsing) are:

- For simplicity of design
- Compiler efficiency is improved
- Compiler portability is enhanced

Tokens, Patterns and Lexemes

- ***Tokens*** are basically a sequence of characters that are treated as a single unit as they cannot be further broken down. It includes keywords (int, float, goto, continue, break etc), identifiers (user-defined names), operators (+,-,/,*), delimiters or punctuations such as comma (,), semicolon (;), braces ({}) etc.
- A ***pattern*** is a set of rules that the scanner or the lexical analyzer follows to create a token. For example, in case of keywords, 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.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Fig 6.2: Examples of tokens

Q. Calculate the number of tokens:

Q1. int a = 10;

Ans: int (keyword), a (identifier), = (operator), 10 (number) and ; (punctuation)

#tokens = 5

Q2.
```
int main()
{
// printf() sends the string inside quotation to
// the standard output (the display)
printf("Welcome to UEC!");
return 0;
}
```

Ans: 'int', 'main', '(', ')', '{', 'printf', '(', ' "Welcome to UEC!" ', ')', ';', 'return', '0', ';', '}'

#tokens = 14

Attributes of Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1. Thus, in many cases the lexical analyzer returns to the parser not only the token name, but also an attribute value that describes the lexeme represented by the token. The most important example is the **token id,** where we need to associate with the token a great deal of information. This information can be its lexeme, its type, the location at which it is first fount etc.

Lexical Errors

Without the help of other components, it is hard for the 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 function identifier. Since **fi** is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phases of the compiler handle the error.

Suppose that a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. Then it should perform some error recovery strategies.

The simplest strategy is **Panic-mode recovery:** We **delete** the successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser.

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 by another character
4. **Transpose** two adjacent characters

6.2 Input Buffering

The task of reading the source program can be speeded up using input buffering. There are many situations where we need to look at least one additional character ahead before we can be sure we have the right lexeme. For instance, we cannot be sure we have seen the end of an identifier until we see a character that is not a letter or a digit. Similarly, in C, single character operators like -, =, <, > can also be the beginning of a two character operators like ==, -=, <=, >=. Thus, a two-buffer scheme is introduced to handle this lookaheads safely.

Both the buffers are of the same size N, and N is usually the size of a disk block. 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 are maintained:

1. Pointer ***lexemeBegin***, marks the beginning of the current lexeme, whose extend we are attempting to determine.
2. Pointer ***forward*** scans ahead until a pattern match is found.

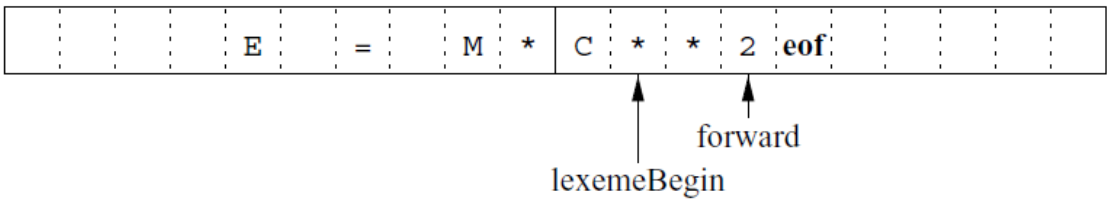


Fig 6.3: Using a pair of input buffers

Once the next lexeme is determined, ***forward*** is set to the character at its right end. Then, after the lexeme is recorded as a token, ***lexemeBegin*** is set to the character immediately after the lexeme just found.

Advancing ***forward*** requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

Sentinels

In the above scheme, at each time we advance ***forward,*** we must check 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: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a special character, known as the **sentinel character**, at the end. Usually we use **eof** as the sentinel character.

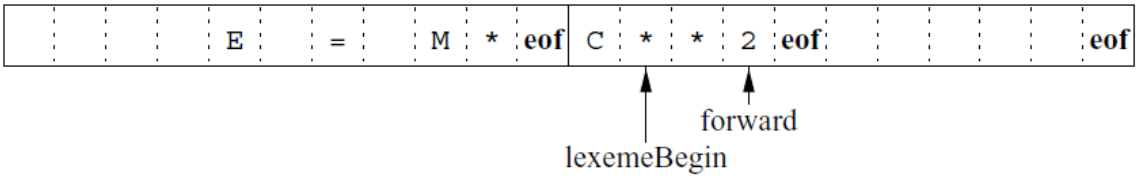


Fig 6.4: Sentinels at the end of each buffer

Figure 6.4 shows the same arrangement of fig 6.3, but with the sentinels added. Any **eof** that appears other than the end of a buffer means the end of the input stream. Fig 6.5 summarizes the algorithm for advancing ***forward*** pointer.

```
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
}
```

Fig 6.5: Lookahead code with sentinels

6.3 Specification of Tokens

Regular expressions are the most important notation for specifying tokens.

Alphabet, Strings and Languages

- An **alphabet** is any finite set of symbols (denoted by Σ). Examples of symbols are letters, digits and punctuations. 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 the string, denoted by $|s|$, is the number of occurrences of symbols in *s*. For example, *banana* is a string of length 6. The ***empty string***, denoted by ε, is the string of length zero.
- A **language** is any countable set of strings over some fixed alphabet.

The following are some string-related terminologies:

1. A ***prefix*** of string *s* is any string obtained by removing zero or more symbols from the end of *s*. For example, *b, ban, banana* and ε are some prefixes of *banana.*
2. A ***suffix*** of string *s* is any string obtained by removing zero or more symbols from the beginning of *s*. For example, *a, nana, banana* and ε are some suffixes of *banana.*
3. A ***substring*** of *s* is obtained by deleting any prefix and any suffix from *s*. For instance, *banana, nan* and ε are some substrings of *banana.*
4. The ***proper*** prefixes, suffixes and substrings of a string *s* are those prefixes, suffixes and substrings respectively of *s*, that are not ε or not equal to *s* itself.
5. A ***subsequence*** of *s* is any string formed by deleting zero or more not necessarily consecutive positions of *s.* For example, *baan* is a subsequence of *banana.*

If x and y are strings, then the **concatenation** of *x* and *y*, denoted *xy*, is the string formed by appending *y* to *x*. For example, if *x* = dog and *y* = house, then *xy* = doghouse. The empty string is the identity under concatenation; that is, for any string *s*, *s*ε = ε*s* = s.

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 ε*s* = s, it follows that s^1 = s. then s^2 = ss, s^3 = sss, and so on.

Operations on Languages

The most important operations on languages are:

- Union
- Concatenation

- Kleene closure
- Positive closure

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	$L^+ = \cup_{i=1}^{\infty} L^i$

Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. 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 L zero times," is defined to be { ε }. Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0. That is, ε will not be in L^+ unless it is in L itself.

Note: Sometimes the empty string, ε, can also be represented using lambda (λ)

Regular Expression

- A language can be finite or infinite.
- All finite languages are accepted by Finite Automata. But an infinite language may or may not be accepted.
- The languages accepted by finite automata are known as Regular Languages.
- Regular expression (RE) is a method to represent regular languages.
- For example, let L be a language such that, L = { ε, a, aa, aaa, aaaa, …. }
  i.e., L is a language consisting of a set of strings with zero or more occurrences of 'a'. This language can be represented using the regular expression, R = a^*
- If *letter* stands for any letter or underscore, and *digit* stands for any digit, then using regular expression we can denote the identifiers in C language as:

$$letter\ (\ letter \mid digit\ )^*$$

- ' | ' denotes union and '*' represents zero or more occurrences
- We can also use '+' to denote union.

Rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote are:

1. 'ε' is a regular expression denoting the set {ε}. i.e., If R = ε, then L(R) = { ε }
2. 'Φ' is a regular expression denoting the set { }. i. e., If R= Φ, then L(R) = { }
3. If '*a*' is a symbol in Σ, then '*a*' is a regular expression, and L(R) = { *a* }

The above three rules are also known as Primitive Rules (i.e. the minimal or basis). In the remaining rules, let's assume that r and **s** are regular expressions denoting the languages L(r**)** and L(**s**) respectively.

1. (r)|(s) is a regular expression denoting the language L(r) U L(s). i.e., the union of two RE is also regular.
2. (r)(s) is a regular expression denoting the language L(r)L(s). i.e., the concatenation of two RE is also regular.
3. Kleene closure of a RE is also regular. i.e., $(r)^*$ is a RE denoting $L((r))^*$
4. If (r) is a RE denoting L(r). i.e., we can add additional pairs of parenthesis around expressions without changing the language they denote.

Some conventions that we adopt in RE are:

1. The unary operator * has highest precedence and is left associative
2. Concatenation has second highest precedence and is left associative
3. Union (| or +) has lowest precedence and is left associative

Under these conventions we may replace the RE (a)|((b)*(c)) by a | b* c. Both these expressions denote the set of strings that are either a single '*a*' or zero or more '*b*'s followed by one '*c*'

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).

Some of the algebraic laws that hold for arbitrary regular expressions r, s and t is given below:

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 distributes over |
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Regular Definitions

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

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

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet Σ U { d_1 , d_2 , d_3 , …., d_{i-1} }

Example 3.5: C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$\begin{aligned} \mathit{letter\_} &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{id} &\rightarrow \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^* \end{aligned}$$

Example 3.6: Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$\begin{aligned} \mathit{digit} &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \mathit{digits} &\rightarrow \mathit{digit} \; \mathit{digit}^* \\ \mathit{optionalFraction} &\rightarrow \texttt{.} \; \mathit{digits} \mid \epsilon \\ \mathit{optionalExponent} &\rightarrow (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\ \mathit{number} &\rightarrow \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent} \end{aligned}$$

Extensions of regular expressions:

1. *One or more instances.* Denoted by + . i.e., a^+ denotes one more instances of *a.*
   $r^* = r^+ \mid \varepsilon$
   $r^+ = r\, r^* = r^* r$

2. *Zero or one instance (?).*
   $r? = r \mid \varepsilon$

3. *Character classes.*
   A regular expression $a_1 \mid a_2 \mid \ldots \mid a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1 a_2 \ldots a_n]$. If $a_1, a_2, \ldots, a_n$ form

a logical sequence, then we replace it by $[a_1 - a_n]$. Thus, [abc] is shorthand for a | b | c, and [a-z] is a shorthand for a | b | c | …. | z.

Example 3.7 : Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$\begin{aligned} letter\_ &\rightarrow [\texttt{A-Za-z\_}] \\ digit &\rightarrow [\texttt{0-9}] \\ id &\rightarrow letter\_ \ (\ letter\_ \mid digit\ )^* \end{aligned}$$

The regular definition of Example 3.6 can also be simplified:

$$\begin{aligned} digit &\rightarrow [\texttt{0-9}] \\ digits &\rightarrow digit^+ \\ number &\rightarrow digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ [\texttt{+-}]?\ digits\ )? \end{aligned}$$

6.4. Recognition of tokens

We use transition diagrams (or Finite Automata) to recognize tokens. Transition diagrams have a collection of nodes or circles, called states. Edges are directed from one state of the transition diagram to another. Each edge is labelled 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* labelled by a (and perhaps by other symbols, as well). 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, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Example 3.9 : Figure 3.13 is a 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.

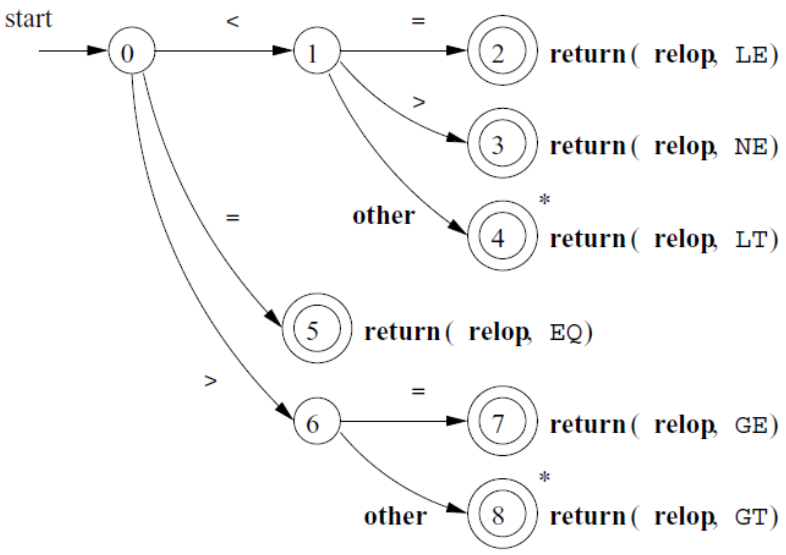


Figure 3.13: Transition diagram for **relop**

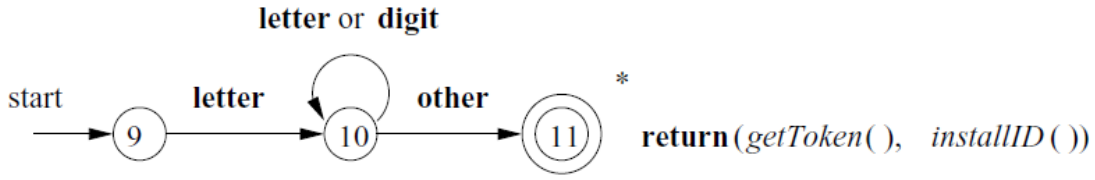


Figure 3.14: A transition diagram for **id**'s and keywords

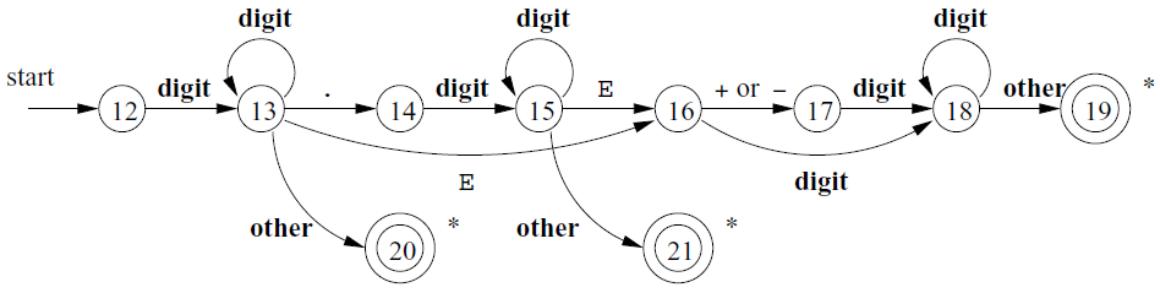


Figure 3.16: A transition diagram for unsigned numbers