Compiler Design

Theme of the subject -"How to design and implement efficient compilers".
We will learn various techniques for implementing the compilers. Efficient compilers assist us to develop software quickly and correctly, meeting all the specified requirements.

UNIT – 1

Introduction: Language Processors, the structure of a compiler, the science of building a compiler, programming language basics.
Lexical Analysis: The Role of the Lexical Analyzer, Input Buffering, Recognition of Tokens, The Lexical-Analyzer Generator Lex, Finite Automata, From Regular Expressions to Automata, Design of a Lexical-Analyzer Generator, Optimization of DFA-Based Pattern Matchers.

1.1 Language Processors or Language Translators:

The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. Before a program can be run, it must be translated into a form, which can be executed by a computer. The software systems that do this translation is called Compiler. Any software which converts one programming language into another programming language is called as Language Translator.
Compiler: -A compiler is a program which takes a program written in a source language and translates it into an equivalent program in a target language. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

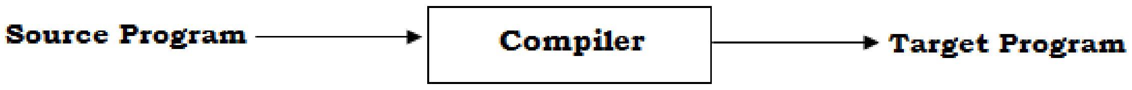


Fig 1.1: Compiler

If the Target Program is an executable machine language program, it can then be called by the user to process input and produce output.

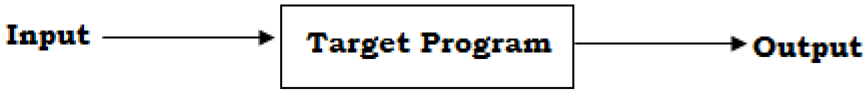


Fig 1.2: Executing the target program

Interpreter:-An interpreter is another common kind of language processor. Instead of producing a target program it directly executes the operations specified in the source program on inputs supplied by the user.

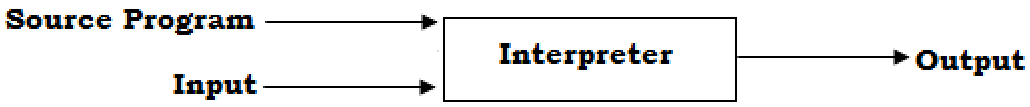


Fig 1.3: Interpreter

❖ ***Note:***

- The machine language target program produced by compiler is much faster than an interpreter at mapping input to output.
- An interpreter gives better error diagnostics than compiler, because it executes the source program statement-by-statement.

Java language processors use both compilation and interpretation shown in figure 1.4. A java program is first compiled to produce intermediate form called bytecode. The bytecode is then interpreted by a virtual machine. The main advantage of this arrangement is that it supports cross platform execution. In order to achieve faster processing of inputs to outputs, JIT (Just in time) compiler is used. It translates bytecode into machine code immediately before running the intermediate program to process the input.

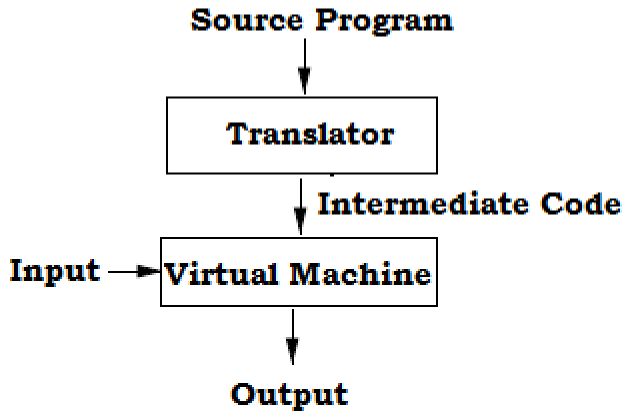


Fig 1.4: Hybrid Compiler

Preprocessor: - A preprocessor is a program that processes its input data (i.e. source program) to produce output (i.e. modified source program) that is used as input to the compiler. The preprocessor expand shorthand's, called macros, into source language statements.

Assembler: If the source program is assembly language and the target language is machine language then the translator is called an assembler.

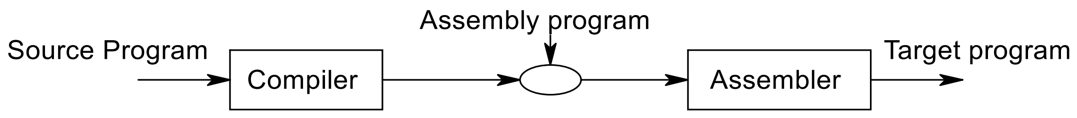


Fig 1.5: Assembler

The assembly language is then processed by a program called on Assembler that produces relocatable machine code as its output.
Reloacatable machine code means that it can be loaded starting at any location ***L*** in memory; i.e., if L is added to all addresses in the code, then all references will be correct.
Linker: Resolves external memory addresses, where the code in one file may refer to a location in another file. It links the relocatable object file with the system wide startup object file and makes an executable file.
Loader: The loader puts together all of the executable object files into memory for execution. It loads the executable code into the memory for execution. The process of loading takes relocatable machine code, alter the relocatable addresses and place the altered instructions and data in memory at the proper locations.

1.2 Language Processing System

The preprocessor may also expand shorthand's, called macros, into source language statements. The modified source program is then fed to a compiler. 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. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

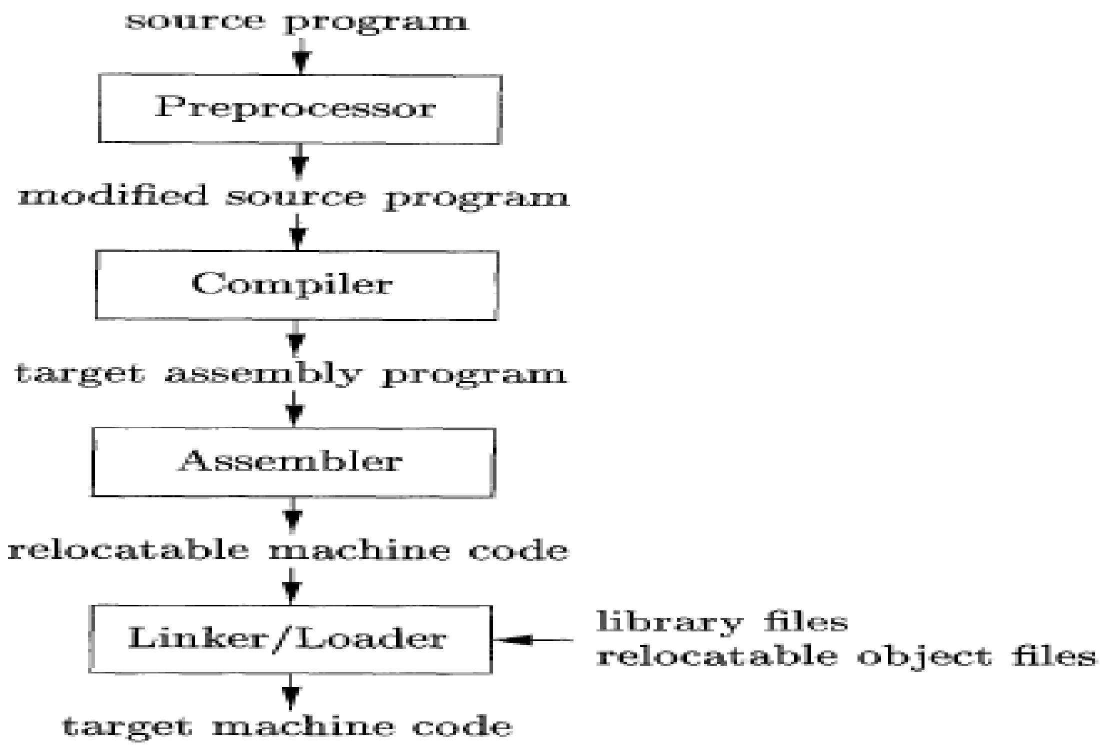


Fig 1.6: Language Processing System

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. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together the entire executable object files into memory for execution.

1.3 The Structure of a Compiler

A compiler maps a source program into semantically equivalent target program. There are two parts to this mapping **Analysis** and **Synthesis.**

(or)

The compilation process can be subdivided into main parts. They are

1. Analysis and 2. Synthesis

Analysis phase: -The analysis part is often called the **front end** of the compiler. In analysis phase, an intermediate representation is created from the given source program. Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and Intermediate Code Generator are the parts of this phase. It breaks up the source program and checks whether the source program is either syntactically or semantically correct. It provides informative error messages, so that the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

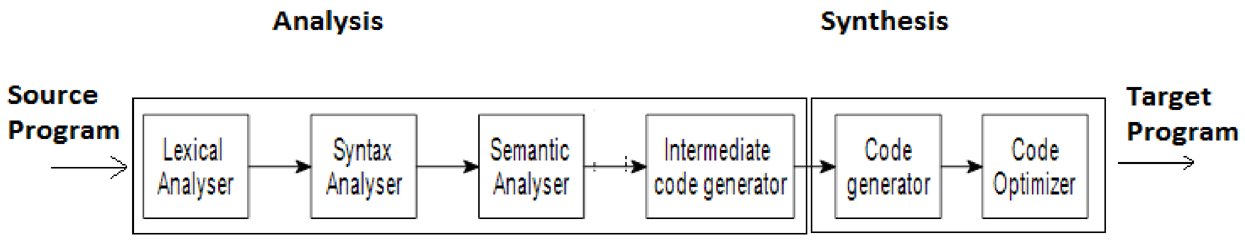


Fig 1.7: Parts of Compiler

Synthesis phase: - The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. Code Generator and Code Optimizer are the parts of this phase. The synthesis part is the **back end** of the compiler. The backend deals with machine-specific details like allocation of registers, number of allowable operators and so on.

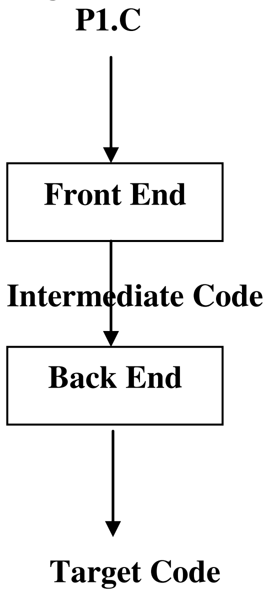


Fig 1.8: Front end and Back end of Compiler

The above fig. shows the two stage design approach of a compiler using C language source File as input.

The main advantages of having this two stage design are as follows:

i. The compiler can be extended to support an additional processor by adding the required back end of the compiler. The existing front end is completely re-used in this case. This is shown in fig below**.**

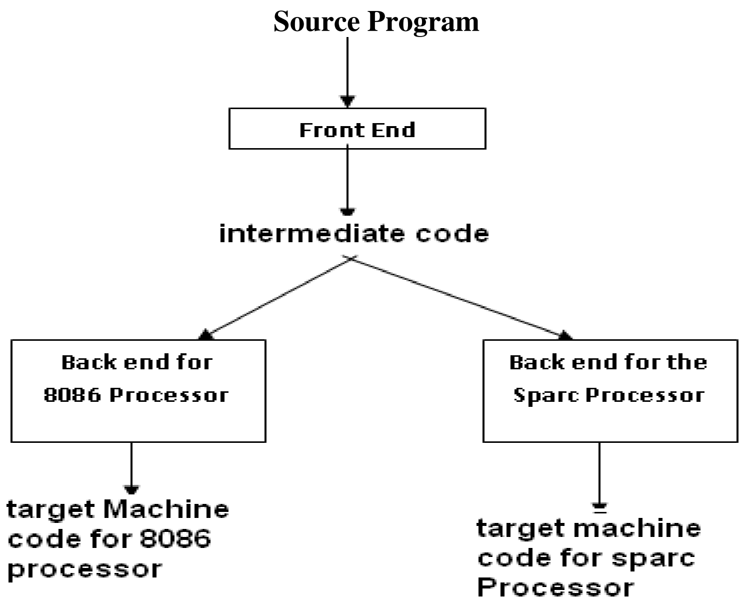

Source Program
Front End
intermediate code
Back end for 8086 Processor
Back end for the Sparc Processor
target Machine code for 8086 processor
target machine code for sparc Processor

Fig.1.9. Supporting an additional processor by adding back end.

ii. The compiler can be easily extended to support an additional input source language by adding required front end. In this case, the back end is completely re-used. This is shown in fig below.

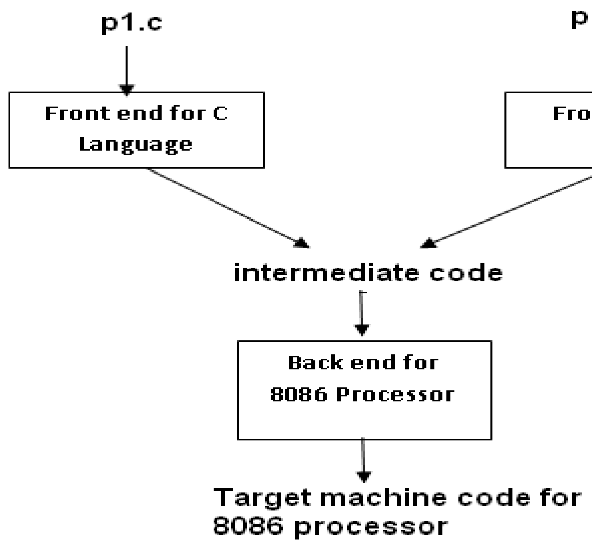


Fig.1.10: Supporting an additional Languages by adding front end.

1.4 Phases of a Compiler

The compilation process operates as a sequence of phases. Each phase transforms the source program from one representation into another representation. A compiler is decomposed into phases as shown in Fig. The symbol table, which stores information about the entire source program, is used by all phases of the compiler. During Compilation process, each phase can encounter errors. The error handler is a data structure that reports the presence of errors clearly and accurately. It specifies how the errors can be recovered quickly to detect subsequent errors.

Lexical Analyzer: -The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the source program and groups the characters into meaningful sequences called lexemes (i.e tokens). For each lexeme, the lexical analyzer produces output in the form

<token-name, attribute-value>

That is passed to the next phase. In 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. Information from the symbol-table entry is needed for semantic analysis and code generator.

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 passed on to the syntax analyzer:

- **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.
- 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. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
- **initial**is a lexeme that is mapped into the token **<id,** 2>, where 2 points to the symbol-table entry for **initial .**
- + is a lexeme that is mapped into the token < + >.
- **rate** is a lexeme that is mapped into the token **<id,** 3>, where 3 points to the symbol-table entry for **rate** .
- * is a lexeme that is mapped into the token <*>.
- 60 is a lexeme that is mapped into the token <60>.

After lexical analysis the assignment statement can be represented as a sequence of tokens as follows:
<id , l >< = ><id, 2><+><id, 3><*><60>
Blanks separating the lexemes would be discarded by the lexical analyzer.
Syntax Analyzer: -The second phase of the compiler is syntax analysis or parsing. A component that performs parsing is called syntax analyzer or parser. The parser uses the tokens produced by the lexical analyzer to create a syntax tree. In syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.
A syntax tree for the token stream is shown as the output of syntax analyzer.
The tree below shows the order in which the operations in assignment are to be performed.

Position=initial + rate * 60

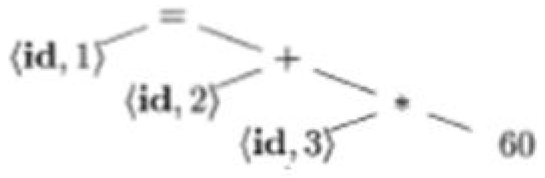


Fig.1.11: Syntax Tree

Semantic Analyzer:- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic errors. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. Some language allows type conversion called coercion. In our assignment statement the type checker converts integer value into floating point number.
The output of the semantic analyzer has an extra node for operator ***inttofloat*** which explicitly converts its integer argument into a floating point number.

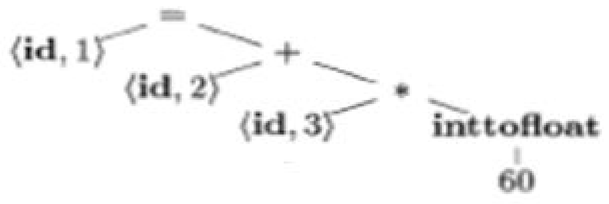


Fig.1.12: Syntax Tree obtained from semantic analyzer

Intermediate Code Generator: - In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an intermediate representation. This intermediate representation should have two important properties:

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

One form of intermediate code is three-address code, which consists of a sequence of assembly-like instructions with atmost three operands per instruction.
The output of Intermediate code generator for our assignment statement consists of Three-address code sequence as follows.
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Code Optimization: -The code-optimization phase is used to produce efficient target code. The target code generated must be executed faster and must consume less power.
Code optimization can also be performed on intermediate code.

- The optimization which is performed on intermediate code is called machine independent code optimization.
- The optimization which is performed on target code is called machine dependent code optimization.

Example***:***

$$\boxed{\begin{array}{c} t1 = id3 * 60.0 \\ id1 = id2 + t1 \end{array}}$$

Code Generator: -The code generator takes intermediate representation of the source program and coverts into the target code. 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 sequences of machine instructions that perform the same task.

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

Symbol Table: -The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
Error Handler: - Error handler should report the presence of an error. It must report the place in the source program where an error is detected. Common programming errors can occur at many different levels.

- Lexical errors include misspellings of identifiers, keywords, or operators.
- Syntax errors include misplaced semicolons or extra or missing braces.
- Semantic errors include type mismatches between operators and operands.
- Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

The main goal of error handler is

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly enough to detect subsequent errors.
3. Add minimal overhead to the processing of correct programs.

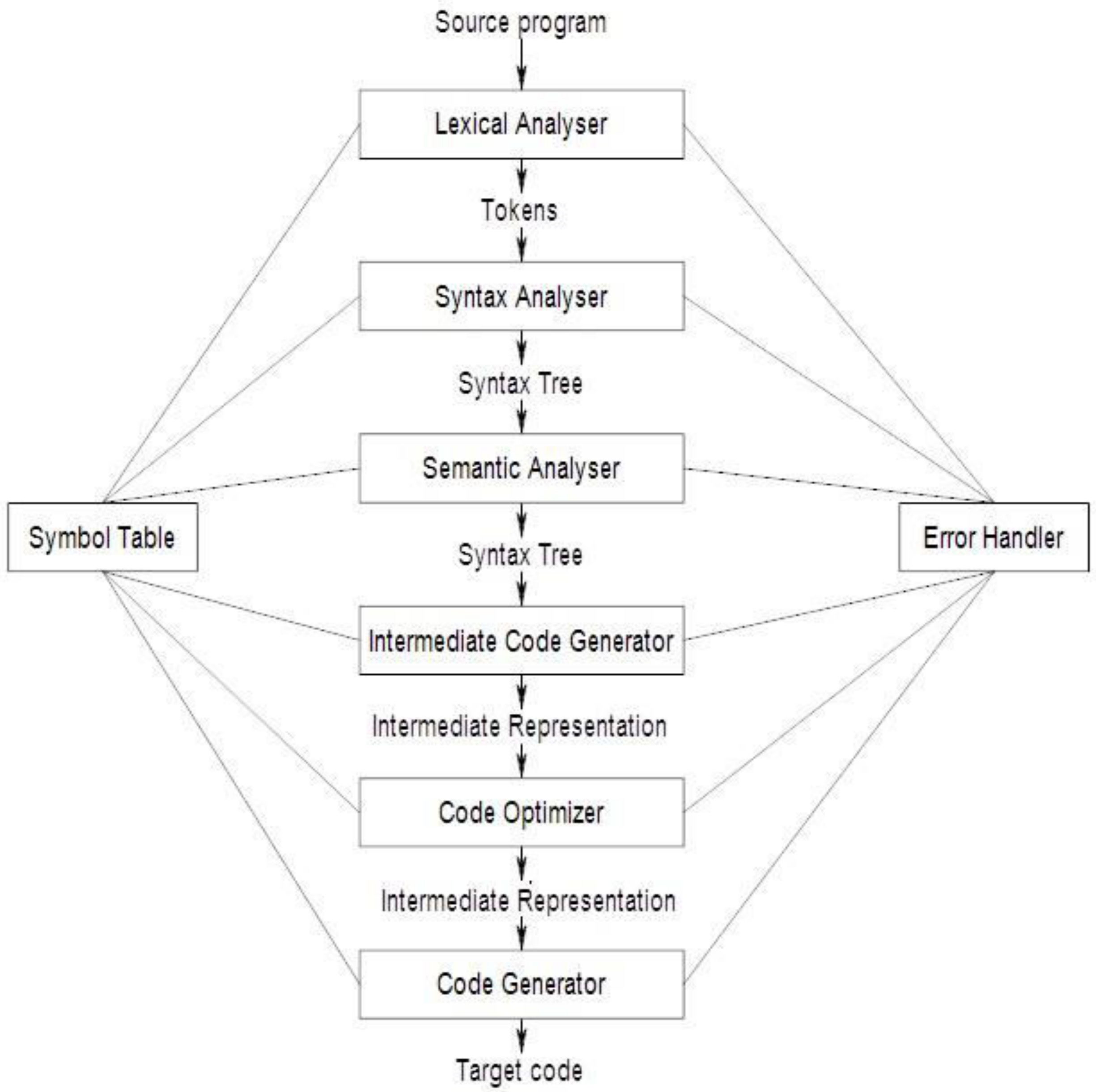
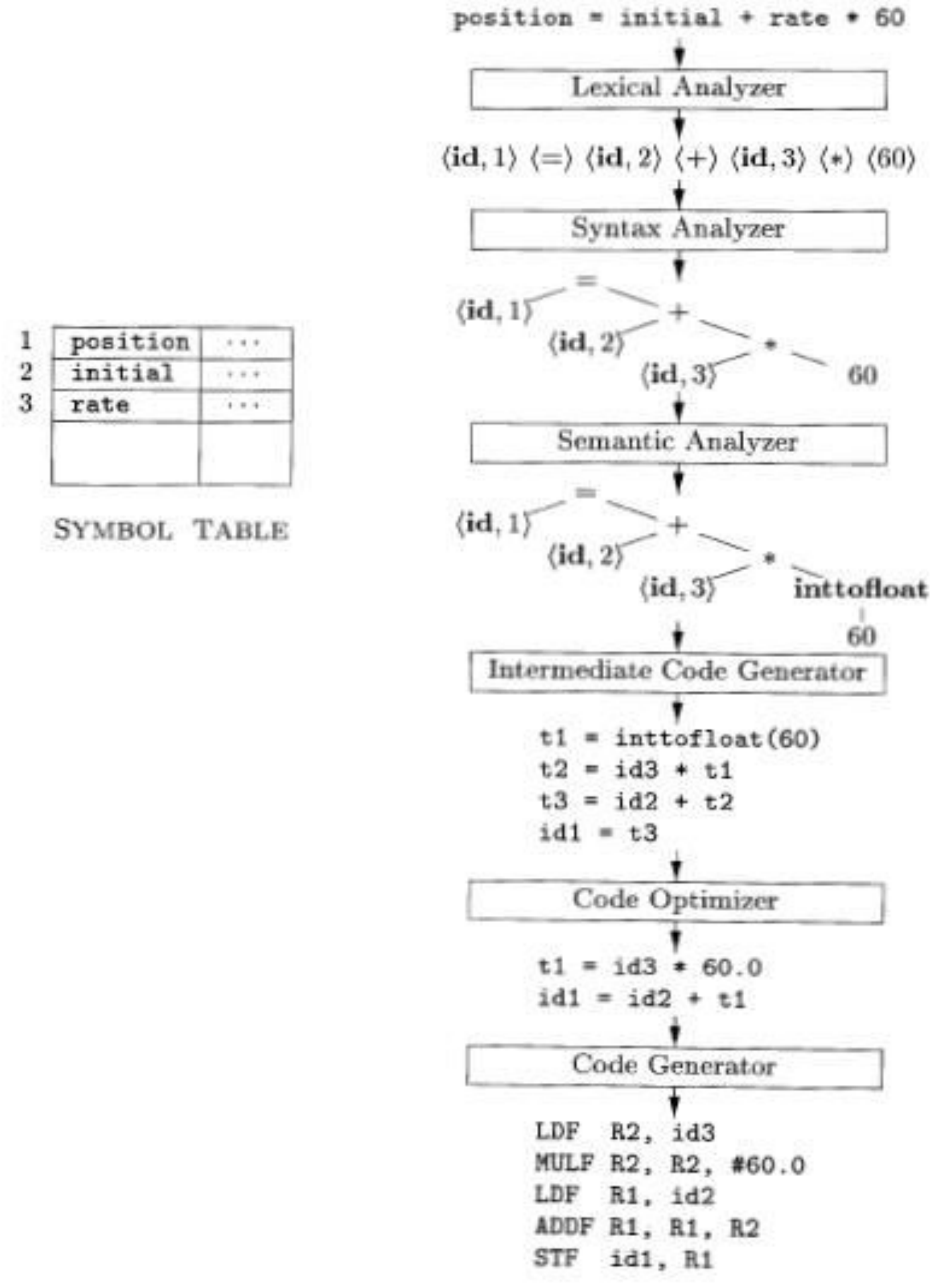


Fig.1.13: - Phases of a compiler

Example: - Compile the statement **position = initial + rate * 60**



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

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

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

Fig.1.14: - Output of each phase of compiler

1.5.Pass and Phase

Phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate

code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.
Multi-pass Compilation requires more memory since we need to store the output of each phase in totality. Multi-pass compiler also takes a longer time to compile since it involves reading of the input in different forms (tokens, parse tree, etc) multiple number of times.
In practice, Compilers are designed with the idea of keeping the number of passes as minimum as possible. The number of passes required in a compiler to process an input source program depends on the structure of programming language. C language compilers can be implemented in a single pass, while ALGOL-68 compilers cannot be implemented in a single pass.

Comparison between Interpreter and Compiler.

S.NO	INTERPRETER	COMPILER
1.	An interpreter is a program which translates each statement of the source program and executes the machine code of that statement.	A Compiler is a program which translates the entire source program and generates the machine code of the source program.
2.	Executing the programs using interpreter is slower.	Executing the programs using compiler is faster.
3.	The source program gets interpreted every time it is to be executed. Hence interpretation is less efficient than compilation.	The source program gets compiled once and the object code of it is stored on the hard disk. It can be used every time the program is to be executed.
4.	Developing interpreter is an easier task.	Developing compiler is a complicated task and is difficult.
5.	Interpreter is simpler and they require less amount of memory.	Compiler is a complex program and it requires large amount of memory.

1.6. The Science of building Compiler

A compiler is a large program which translates high level language into an equivalent machine code. A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled.

i) Modeling in compiler design and implementation

Implementation of compiler involves the design of right mathematical models and selection of right algorithms. Some of most fundamental models are finite-state machines and regular expressions. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages.

ii) The Science of Code Optimization

Code Optimization- It is a program transformation technique which improves the code such that the resultant target code will get execute faster by consuming less resource.
The optimization technique should not influence the semantics of the input source program. There are number of code optimization techniques that can be applied for developing the efficient compilers. Code optimization schemes must meet the following design objectives.

Compiler optimization must meet the following ***design objectives***:

- Optimization must preserve the meaning of the compiled program.
- Optimization must improve the performance.
- Compilation time must be kept minimum.
- The engineering effort required must be manageable.

There are an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations and implement only those that lead to the better performance of the code.

1.7. Programming Language Basics

i) ***Static scope and dynamic scope***

Scope refers to a place in a program where a variable is visible and can be referenced.

The scope of a declaration of **x** is the region of the program in which uses of x refer to this declaration.

A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

Example: ***public static int x;***

The above declaration makes **x** a class variable and says that there is only one copy of **x**, no matter how many objects of this class are created. Moreover, the compiler can determine allocation in memory where this integer x will be held. If "static" is omitted from this declaration, then each object of the class would have its own location where x would be held, and the compiler could not determine all these places in advance of running the program.

ii) ***Environments and State***

The *environment* is a mapping from names to locations in memory. Since variables refer to locations ("l-values" in the terminology of C), we could alternatively define an environment as a mapping from names to variables. The *state* is a mapping from locations in store to their values. That is, the state maps l-values to their corresponding r-values, in the terminology ofC. Environments change according to the scope rules of a language.

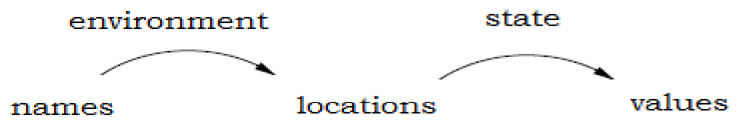


Fig.1.15: Two-Stage mappings from names to values

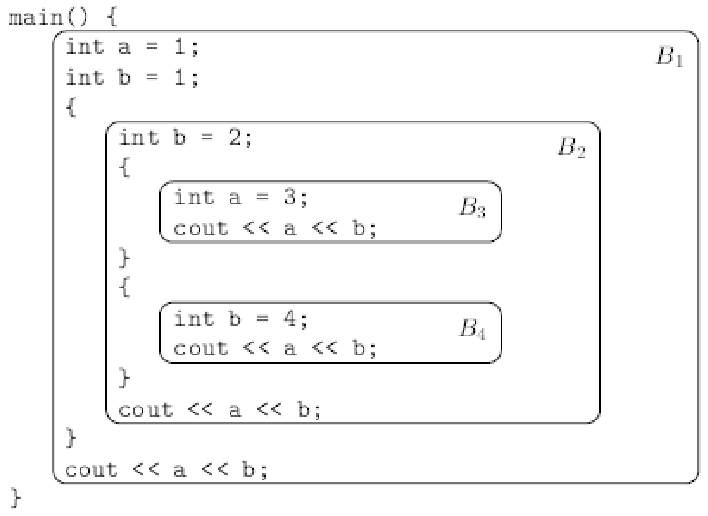
- Binding of names to locations is dynamic.
- The binding of locations to values is dynamic, since we cannot determine the value in a location until we run the program.

iii) ***Static Scope and Block Structure***

Most languages, including C and its family, use static scope. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like public, private, and protected. In section explains static-scope rules for a language with blocks. A block is a grouping of declarations and statements. C uses braces { and} to delimit a block; Algol uses begin and end for the same purpose.

We say that a declaration D "belongs" to a block B if B is the most closely nested block containing D; that is, D is located within B, but not within any block that is nested within B.

The static-scope rule for variable declarations in block-structured languages is as follows. If declaration D of name x belongs to block B, then the scope of D is all of B, except for any blocks B' nested to any depth within B, in which x is redeclared. Here, x is redeclared in B' if some other declaration D' of the same name x belongs to B'



```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Consider the declaration int **a = 1**in block B1. Its scope is all of B1, except for those blocks nested (perhaps deeply) within B1 that have their own declaration of **a**. B2, nested immediately within B1, does not have a declaration of **a**, but B3 does. B4 does not have a declaration of **a**, so block B3 is the only place in the entire program that is outside the scope of the declaration of the name **a** that belongs to B1. That is, this scope includes B4 and all of B2 except for the part of B2 that is within B3.

DECLARATION	SCOPE
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Table 1.1: Scopes of declaration for above example

iv) Explicit Access Control

Classes and structures introduce a new scope for their members. If p is an object of a class with a member x, then the use of x in p.x refers to field x in the class definition. The scope of a member declaration x in a class C extends to any subclass C', except if C' has a local declaration of the same name x.

The usage of keywords like public, private, and protected, object-oriented languages such as C++ or Java provide explicit control over access to member names in a super class. These keywords support encapsulation by restricting access.

The private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

v) Dynamic Scope

The term dynamic scope, usually refers to the following policy: a use of a name **x** refers to the declaration of **x** in the most recently called, not-yet-terminated, procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object oriented programming.

vi) Parameter Passing Mechanisms

There are two types of parameters actual parameters (the parameters used in the call of a procedure)are associated with the formal parameters (those used in the procedure definition). The most common parameter passing mechanisms are:

a) Call by Value

In this parameter passing mechanism, the changes made to formal parameters will not be reflected on the actual parameters because both actual and formal parameters have separate storage locations.

b) Call by reference

In this parameter passing mechanism, the changes made to formal parameters will be reflected on the actual parameters because formal parameters references the storage locations of actual parameters. Hence, the any change made on formal parameter will be reflected on actual parameter.

c) Call by Name

This technique was used in early programming language such as Algol. In this technique, symbolic "name" of a variable is passed, which allows it both to be accessed and update.. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee.

Consider the example below:

```
procedure double(x);
        real x;
begin
        x:=x*2
end;
```

In general, the effect of pass-by-name is to substitute the argument expression in a procedure call for the corresponding parameters in the body of the procedure, e.g. double(c[j]) is interpreted as c[j]:=c[j]*2.

vii) Aliasing

It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. Suppose *a* is an array belonging to a procedure *p,* and *p* calls another procedure *q(x,y)* with a call *q(a,a).* Now, *x* and *y* have become aliases of each other.

************Important Questions************

1. Define language processor? Differentiate between Compiler and Interpreter.
2. Explain in detail about various Phases of Compiler.

   (Or)

   Explain the different Phases of a Compiler, showing the output of each phase for the statement "**position=initial+rate*60**".

   ii) **x=(a+b)*(c+d)**

   iii) **Fahrenheit=Celsius*1.8+32**
3. Differentiate between Pass and Phase.
4. Explain about Language Processing System.
5. Define Loader, Linker and Assembler. Explain the structure of a Compiler & its advantages.
6. Describe various parameter passing techniques.

Lexical Analysis – Role of Lexical Analysis – Lexical Analysis Vs Parsing – Token, patterns and Lexemes – Input Buffering - Lexical Errors - Regular Expressions – Regular definitions Recognition of Tokens- Lexical Analyzer Generator (LEX Tool).

1. *Introduction:*

In this chapter we will discuss how to construct a lexical analyzer.
There are three ways to implement Lexical Analyzer:

i) Using State Transition diagrams to recognize various tokens.
ii) We can write a code to identify each occurrences of each lexeme on the input and to return information about the token identified.
iii) We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical analyzer generator and compiling those patterns into code that functions as lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patters, not the entire program. Lexical analyzer generator called **LEX.**

Lexical Analysis: - The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the source program and groups the characters into meaningful sequences called lexemes. It identifies the category (i.e tokens) to which this lexeme belongs. For each lexeme, the lexical analyzer produces output in the form

<token-name, attribute-value>

This output is passed to the subsequent phase i.e syntax analysis.

2. *Role of the Lexical Analyzer*

Lexical analyzer is the first phase of a compiler. The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. When lexical analyzer discovers a lexeme constituting an identifier, it interacts with the symbol table to enter that lexeme into the symbol table. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The getNextToken command given by the parser, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce the next token, which it returns to the parser.

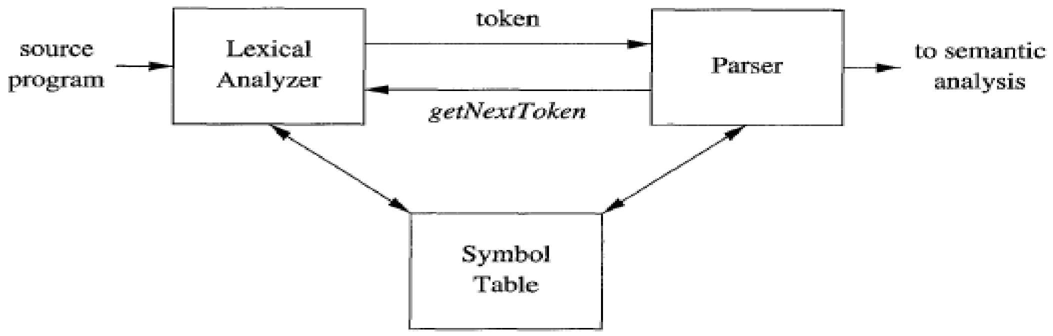


Fig. 1.2.1: Role of lexical analyzer.

Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. These tasks are as follows:

- Stripping out comments and whitespaces.
- Correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.

Sometimes, lexical analyzers are divided into two processes:

a. **Scanning** consists of the simple processes that perform such as deletion of comments and eliminating excessive whitespace characters into one.
b. **Lexical analysis** is the more complex portion, which produces the sequence of tokens as output.

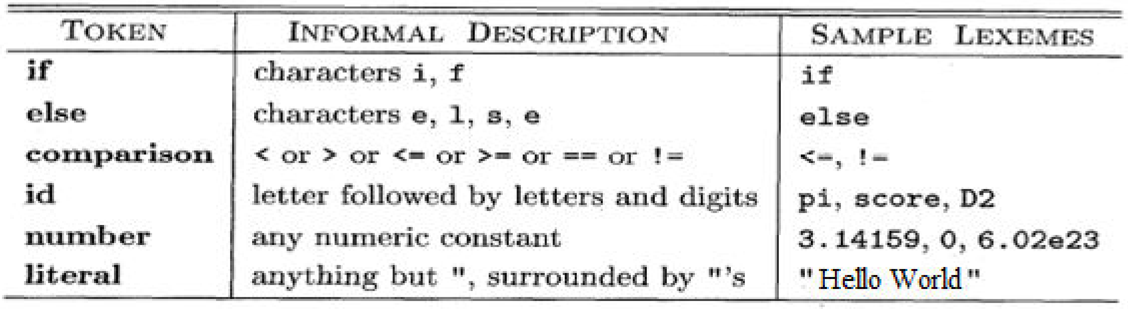
3. Lexical Analysis Vs. Parsing

There are a number of reasons for separating Lexical Analysis and Parsing.

i) To simplify the overall design of the compiler.
ii) Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
iii) Compiler portability is enhanced.

4. Token, patterns and Lexemes: -

- A token is a sequence of characters having a collective meaning.
  A **token** is a pair consisting of a token name and an optional attribute value. The token name is the category of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier etc.
- A **pattern** is a description that specify the rules that the lexemes should follow in order to belong to that token.
- 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.
  Example : printf("total = %d\n", score);
  printf and ***score*** are lexemes matching the pattern of token **ID**.



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	"Hello World"

In many programming languages, the following classes cover most or all of the tokens:

i) One token for each keyword. The pattern for a keyword is the same as the keyword itself.
ii) Tokens for the operators, either individually or in classes such as the token comparison mentioned.
iii) One token representing all identifiers.
iv) One or more tokens representing constants, such as numbers and literal strings.
v) Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

5. Input Buffering

To recognize the right lexeme we have to look one or more characters beyond the next lexeme. For example we cannot be sure that we have seen the end of identifier until we see the character that is not a letter or digit and therefore not a part of the lexeme id. The input character is read from secondary storage, but reading in this way from secondary storage is costly. To reduce the input processing time two buffer scheme is introduced. This scheme has two buffers that are alternatively reloaded as shown in figure below.

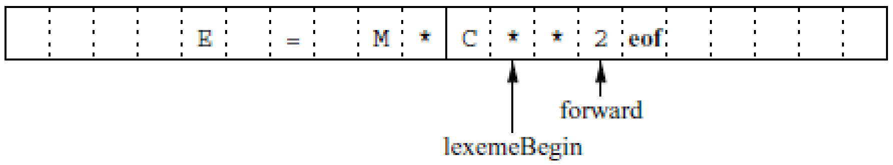


Fig.1.2.2: Using pair of input buffer

Buffer Pairs (or) Two Buffer Scheme

In this method two buffers are used to store the input string. Each buffer is of the same size *N,* and *N* is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into buffer, rather than using one system call per character. If fewer than N characters remain in input file, then **eof** character is used to mark the end of file.

Two pointers to input are maintained.

i. **lexemeBegin** pointer marks the beginning of the current lexeme.
ii. **Forward pointer** scans ahead until a pattern match is found.

Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recognized, an attribute value of a token is returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme just found. In fig 1.2.2, we see forward is set to the character immediately after lexeme just found.

The first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. 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. By using this scheme we must check 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: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a *sentinel* character at the end. The sentinel is a special character that cannot be part of the source program, **eof** character is considered as sentinel**.** The usage of sentinels is shown in figure below. Note that **eof** retain its use 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.

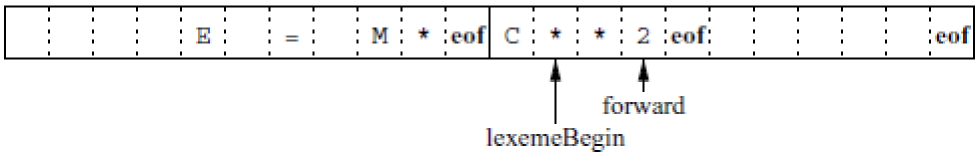


Fig.1.2.3: Sentinels at the end of each buffer

6. Lexical Errors

Lexical errors include misspellings of identifiers, keywords, or operators. It is hard for a lexical analyzer to tell, without the help of other components, that there is a source-code error. For instance, if the string **fi** i.e fi( ) is encountered for the first time in a C program in the context 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 the parser handle an error due to transposition of the letters. However, suppose a situation arises where the lexical analyzer is unable to proceed because the lexeme doesn't matches any of the patterns for tokens. The simplest recovery strategy is ***"panic mode" recovery.*** We delete 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, but in an interactive computing environment it may be quite adequate. Other possible error-recovery actions are:

i) Delete one character from the remaining input.
ii) Insert a missing character into the remaining input.
iii) Replace a character by another character.
iv) Transpose two adjacent characters.

7. Regular Expressions

A regular expression is a pattern that describes a set of strings.

Regular expressions are used to describe the languages.

The regular expression is the one that matches a single character. For example: 's' matches any input string where letter ***s*** is present like ***sink, base***.

Meta characters in Regular Expression:

. → matches any character except a new line.
^ → matches the start of the line.
$ → matches end – of- the – line.
[ ] → A character class- matches any letter within the parenthesis. [0123456789] matches 0 or 1 or 2 etc.
[^ abcd] → ^ inside the square bracket represents the match of any character except the ones in the bracket.
| → matches either preceding Regular Expression or Succeeding Regular Expression.
( ) → used for grouping Regular Expressions
***,+,?** → for specifying repetitions in Regular Expressions
***** zero or more occurances
+ one or more occurrences
? zero or one occurrences
{ } → indicates how many times the previous pattern is matched. Eg. A {1,3} represents a match of one to three occurrences of 'a'. The strings that matches are 'dad','daad','daaad' etc.
Regular expressions are an important notation for specifying lexeme patterns. To describe the set of valid C identifiers use a notation called regular expressions. In this notation, if letter_ is established to stand for any letter or the underscore, and digit is established to stand for any digit. Then the identifiers of C language are defined by

letter (letter| digit)*

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of". The letter at the beginning indicates that the identifier can contain any letter at the beginning. The regular expressions are built recursively out of smaller regular expressions.
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),* which is also defined recursively from the languages denoted by r's subexpressions. 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. ∈ (epsilon) is a regular expression, and *L(∈)* is {∈}, that is, the language whose 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.
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 highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative.
There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure below shows some of the algebraic laws that hold for arbitrary regular expressions *r, s,* and *t.*

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

8. Regular definitions

Regular Definitions are names given to certain regular expressions and those names can be used in subsequent expressions as symbols of the language. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

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

where

1. Each $\mathbf{d_i}$ is a new symbol, not in Σ and not the same as any other of the $\mathbf{d_i}$'s, and
2. Each $\mathbf{r_i}$ is a regular expression over the alphabet Σ U $\{d_l, d_2, \ldots, d_{i-l}\}$.

Example 1 **:** C identifiers are strings of letters, digits, and underscores. Write a regular definition for the language of C identifiers.

$$\begin{array}{rcl} \textit{letter} & \rightarrow & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \mid \\ \textit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \textit{id} & \rightarrow & \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\ )^* \end{array}$$

Using shorthand notations, the regular definition can be rewritten as:

$$\begin{array}{rcl} \textit{letter} & \rightarrow & [\mathtt{A\text{-}Za\text{-}z\_}] \\ \textit{digit} & \rightarrow & [\mathtt{0\text{-}9}] \\ \textit{id} & \rightarrow & \textit{letter}\ (\textit{letter} \mid \textit{digit}\ )^* \end{array}$$

Example 2 **:** Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. Write a regular definition for unsigned numbers in C language.

$$\begin{array}{rcl} \textit{digit} & \rightarrow & [\mathtt{0-9}] \\ \textit{digits} & \rightarrow & \textit{digit}^+ \\ \textit{number} & \rightarrow & \textit{digits}\ (\ .\ \ \textit{digits})?\ (\ \mathtt{E}\ [+-]?\ \ \textit{digits}\ )? \end{array}$$

9. Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we study how to take the 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.

The below example describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

```
stmt   —›  if expr then stmt
        |  if expr then stmt else stmt
           e
expr   —›  term relop term
        |   term
term   —›  id
        |   number
```

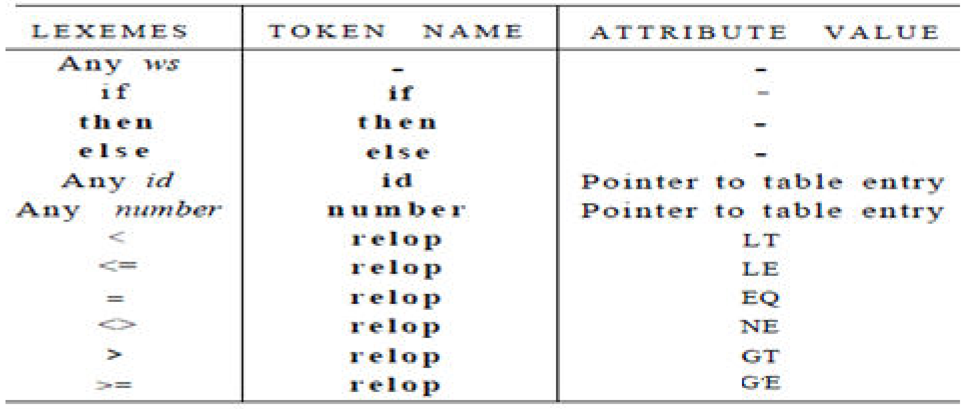
The terminals of the grammar, which are **if, then, else, relop, id,** and **number,** are the names of tokens. The patterns for these tokens are described using regular definitions.

```
   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   ---->  < | > | <= | >= | <> | =
```

For this language, the lexical analyzer will recognize the keywords **if**, **then,** and **e l s e ,** as well as lexemes that match the patterns for *relop, id,* and *number.* To simplify matters, we make the common assumption that keywords are also *reserved words:* that is, they are not identifiers, even though their lexemes match the pattern for identifiers.
In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

ws → (tab|blank space|new line)

Here, **blank, tab,** and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. The table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value is returned.



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

Transition Diagrams

Compiler converts regular-expression patterns to 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. 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 (and perhaps by other symbols, as well). If we find such an edge, we enter the state of the transition diagram to which that edge leads.
Some important conventions about transition diagrams are:

i) 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.

ii) 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.

iii) 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.

Below 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. On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5. The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character).

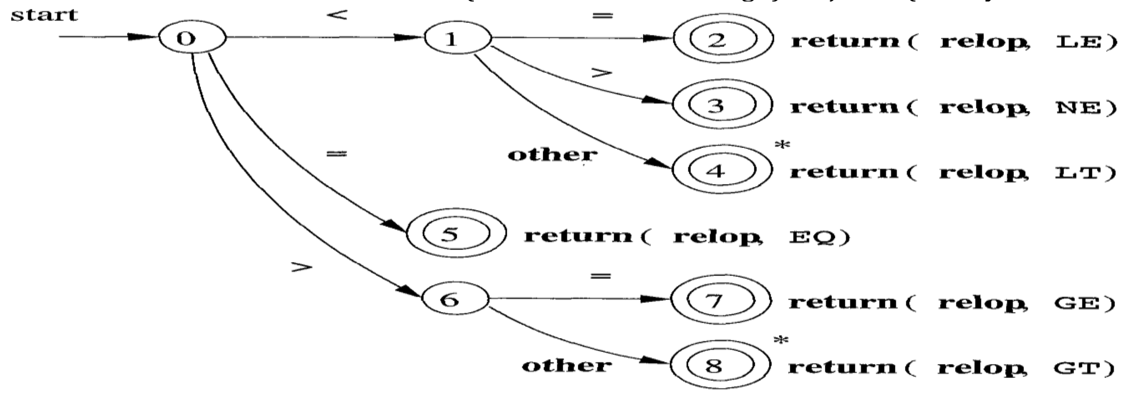


Figure: Transition diagram for relop

10. Recognition of Reserved Words and Identifiers

Recognizing keywords and identifiers presents a problem. Usually, keywords like **if** or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. The below diagram will recognize the keywords **if, then,** and **e l s e** of our running example.

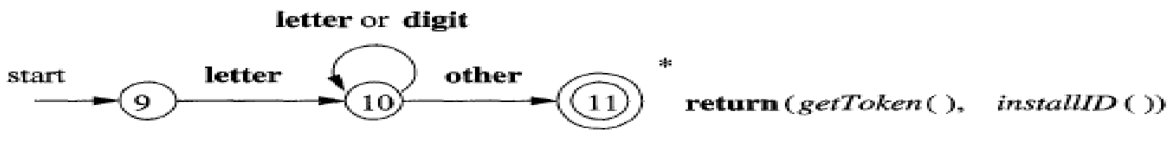


Figure: Transition diagram for identifier

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

i) 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. When we find an identifier, a call to *installlD* 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. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id.** The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

ii) Create separate transition diagrams for each keyword; an example for the keyword **then** is shown in Fig.

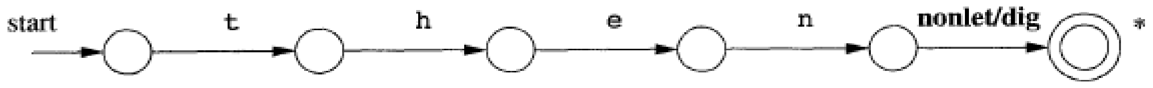


Figure: Transition diagram for then

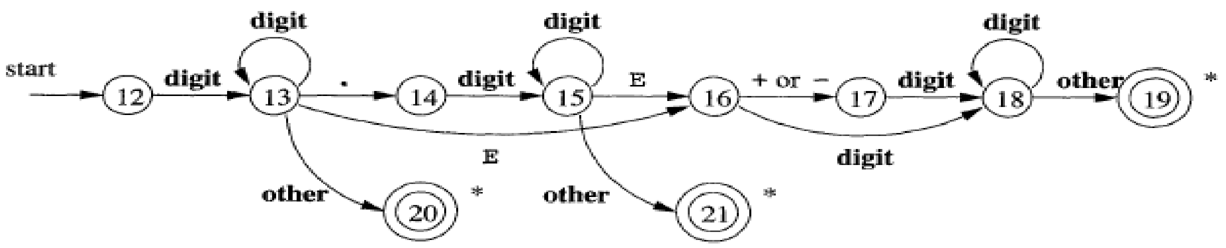


Figure: Transition diagram for unsigned numbers

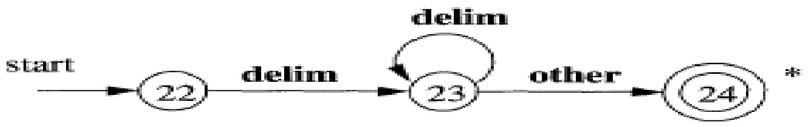


Figure: Transition diagram for whitespace

11. LEXICAL ANALYZER GENERATOR LEX

Lex is a tool or in a more recent implementation Flex, that allows us to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler.* Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called ***lex.yy.c*** that simulates this transition diagram.

USE OF LEX:

Figure below shows how LEX is used. An input file, which we call ***lex.l*** , is written in the Lex language and describes the lexical analyzer to be generated. This file is given as input to the LEX Compiler. The LEX compiler transforms ***lex.l*** to a C program, in a file that is always named ***lex.yy.c.*** The latter 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.

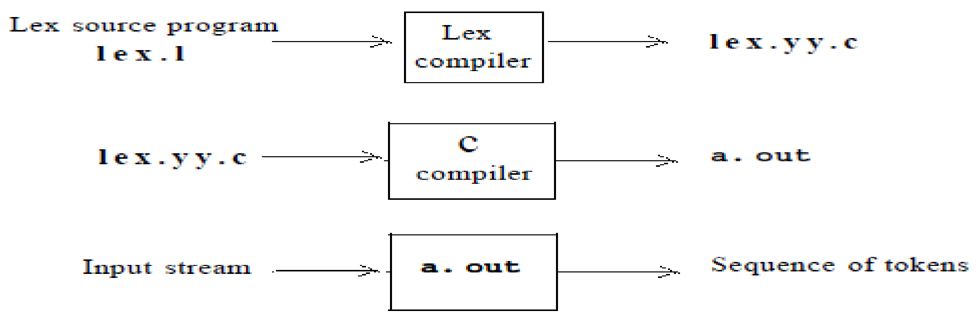


Figure Creating a lexical analyzer with **Lex**

Structure of LEX Program:

The LEX program consists of the following sections.

```
declarations
 %%
translation  rules
 %%
auxiliary  functions
```

• **Declaration Section:** Consists of regular definitions that can be used in translation rules.
Example: letter {a-zA-Z}
Apart from the regular definitions, the declaration section usually contains the # defines, C prototype declaration of functions used in translation rules and some # include statements for C library functions used in translation rules. all these statements are mentioned between special brackets %{ and %}.
Example: *%{*
# define WORD 1
%}
These statements are copied into **lex.yy.c.**
• **Translation Rules Section**: consists of statements in the following form

Pattern 1 { Action 1 }
Pattern 2 { Action 2 }
....
Pattern N { Action N }

Each pattern is a regular expression, which may use the regular definitions of the declaration section. Where Pattern 1, Pattern 2,…,Pattern N are regular expressions and the Action 1,Action 2,…Action N are all program segments describing the action to be taken when the pattern matches.

- **Auxiliary Functions section:** usually contains the definition of the C functions used in the action statements. The whole section is copied as is into **lex.yy.c.** These functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns *Pi.* It then executes the associated action *Ai.* Typically, *Ai* will return to the parser, but if it does not (e.g., because *Pi* describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable ***yylval*** to pass additional information about the lexeme found, if needed.
The actions taken when ***id*** is matched are listed below:

1. Function ***installID()*** is called to place the lexeme found in the symbol table.

2. This function returns a pointer to the symbol table, which is placed in global variable ***yylval*** , where it can be used by the parser or a later component of the compiler. Note that ***installID()*** has available to it two variables that are set automatically by the lexical analyzer:
   (a) ***yytext*** is a pointer to the beginning of the lexeme.
   (b) ***yyleng*** is the length of the lexeme found.
3. The token name **ID** is returned to the parser.
   The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function installNum0.

```
%{
     /* definitions of manifest constants
     LT, LE, EQ, NE, GT, GE,
     IF, THEN, ELSE, ID, NUMBER, RELOP */
 '/.}

 /* regular definitions */
 delim       [ \t\n]
 ws          {delim}+
 letter       [A-Za-z]
 digit        [0-9]
 id          {letter}({letter}|{digit})*
 number       {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
 %%
 {ws}        {/* no action and no return */}
 if          {return(IF);}
 then        {return(THEN);}
 else        {return(ELSE);}
 {id}        {yylval = (int) installID(); return(ID);}
 {number}    {yylval = (int) installNumo ; return(NUMBER);}
  "<"        {yylval = LT; return(RELOP) ;}
 "<="        {yylval = LE; return(RELOP) ;}
  "="        {yylval = EQ; return(RELOP) ;}
 "< >"       {yylval = NE; return(RELOP) ;}
 " >"        {yylval = GT; return(RELOP) ;}
 ">="        {yylval = GE; return(RELOP) ;}
  %%
  int  installID()    {/* function to install the lexeme, whose
                          first character is pointed to by yytext
                          arid whose length is yyleng, into the
                          symbol table and return a pointer

                         thereto */}
int installNumo {/* similar to installlD, but puts numer-
                       ical constants into a separate table

                    */}
```

Example: Write a LEX Program to recognize tokens in the given arithmetic expression.

a=b+c*10;

```
%{
#include<stdion.h>
%}
letter [a-zA-Z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?
%%
{id} {printf("%s is an Identifier\n", yytext);}
{num} {printf("%s is a Number\n", yytext);}
 "+" {printf("%s is an Arithmetic Operator\n", yytext);}
 "-" {printf("%s is an Arithmetic Operator\n", yytext);}
"*" {printf("%s is an Arithmetic Operator\n", yytext);}
"/" {printf("%s is an Arithmetic Operator\n", yytext);}
"=" {printf("%s is an Assignment Operator\n", yytext);}
";" {printf("%s is a Punctuation\n", yytext);}
%%
main()
{
  yylex();  /* to invoke lexical analyzer */
}
 yywrap()
{
 return 1; /* returns 1 when the end of input is found */
}
```

Compilation & Execution

```
$ lex tokens.l
$ cc lex.yy.c
$ ./a.out
```

Input: a=b+c*10;

```
^D
```

Output:

```
a is an Identifier
= is an Assignment Operator
b is an Identifier
+ is an Arithmetic Operator
c is an Identifier
* is an Arithmetic Operator
10 is a Number
; is a Punctuation
```

Finite Automata, From Regular Expressions to Automata, Design of a Lexical-Analyzer Generator, Optimization of DFA-Based Pattern Matchers.

1.3.1. Finite Automata

- Finite Automaton: It is an abstract machine (or) mathematical model of a computer.
- It is a simple machine to recognize patterns.
- Finite automata are recognizers: for given input string, finite automata produce output as either yes or no.

F.A comes in two flavors:

a) **Non-Deterministic Finite Automata (NFA):** it is a finite state machine in which there may be more than one transitions from present state on given input. NFA has a power to be in several states at once.
A NFA is expressed mathematically by using quintuple notation as:
M = (Q, ∑, q_0, F, δ)
Where
Q **:** is a finite set of states
∑ **:** is a finite input alphabet
q_0 **:**is a initial state, $q_0 \in Q$
F *: is a set of final states,* $F \subseteq Q$
δ: Q × Σ → 2^Q ***(Powerset of Q)***

b) **Deterministic Finite Automata (DFA):** for every pair of state and input symbol there is a unique next state.
DFA is a special case of NFA where:
i) There are on moves on input ε.
ii) For each state ***s*** and input symbol ***x***, there is exactly one edge out of ***s*** labelled ***x***.
A DFA is mathematically represented by using **Quintuple (Five) Notation** as
M = (Q, ∑, δ, q_0 , *F*)
Where
Q **:** is a finite set of states.
∑ **:** is a finite set of input symbols.
δ: is a transition function which maps current state and current input symbol to produce next state. **δ: Q × Σ → Q**
q_0 **:** initial state, $q_0 \in Q$.
F *: Set of final states,* $F \subseteq Q$.

Every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.
Example: NFA to recognize the language of regular expression **(a|b)*abb**.

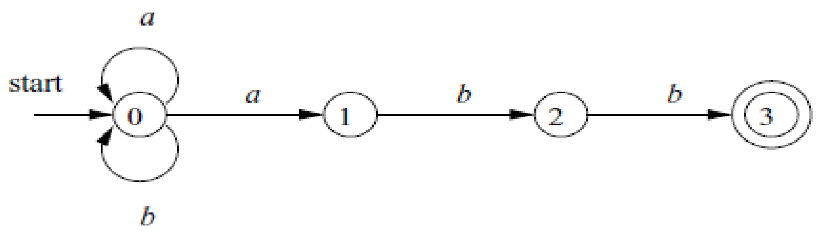


Fig: NFA accepting (a|b)*abb.

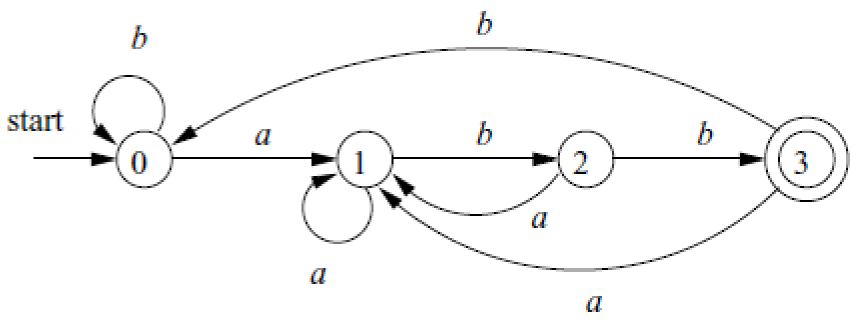


Fig: DFA accepting (a|b)*abb.

1.3.2. Simulating a DFA

The above DFA can be simulated by writing the following algorithm.

Input: an input string x terminated by eof character. A DFA start state is S_0, Accepting states F and transition function move(s,c).

Output: Answer either "Yes" or "No".

Method: Apply the following algorithm on input string x. The move(s,c) gives the state transition from s on input symbol c. The function nextChar( ) returns the next character of the input string x.

```
s=s0
c=nextChar();
while(c!=eof)
{
s=move(s,c);
c=nextChar();
}
if(s is in F)
return "Yes";
else
return "No";
```

1.3.3. From Regular Expressions to Automata:

Regular expressions are used to describe the patterns of tokens that are recognized by lexical analyzer.

Conversion of NFA-ε to DFA:

Converting a given NFA-ε into an equivalent DFA increases the number of states in DFA to atmost 2^n.

Consider the given NFA-ε is represented as **$N=(Q,\Sigma,\delta',q_0,F)$.** The resultant DFA after conversion is represented as **$D=(Q_D,\Sigma,\delta_D,[q_0],F_D)$.**

1. Compute **ε-closure** of start state of NFA. The result of ε-closure gives the start state of DFA. The set of states of DFA is **Q_D.**
2. While (there are unmarked states **T** in **Q_D**)
3. {
4. Mark **T**;
5. For (each input symbol **a**)
6. {
7. **$\delta_D([T,a])=$ ε-closure$(\delta'(T,a))$**

8. If the obtained state from step 7 is the new state, consider it as unmarked and add it to the **Q_D**.
9. }
10. }
11. Mark a state as final state if it includes any final state(s) of NFA.

Problem: Construct the DFA for the given NFA-ε.

	a = 0	a = 1	a = 2	a = ε
$\rightarrow q_0$	q_0	Ø	Ø	q_1
q_1	Ø	q_1	Ø	q_2
$*q_2$	Ø	Ø	q_2	Ø

Solution: compute ε-closure(q_0)={q_0, q_1 , q_2}.
Mark [q_0, q_1 , q_2] as start state of DFA.

$$\begin{aligned}\delta_D([q_0, q_1, q_2], 0) &= \varepsilon\text{-closure}(\delta'([q_0, q_1, q_2], 0))\\ &= \varepsilon\text{-closure}(\delta'(q_0, 0) \cup \delta'(q_1, 0) \cup \delta'(q_2, 0))\\ &= \varepsilon\text{-closure}(\{q_0\} \cup \{Ø\} \cup \{Ø\})\\ &= [q_0, q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta_D([q_0, q_1, q_2], 1) &= \varepsilon\text{-closure}(\delta'([q_0, q_1, q_2], 1))\\ &= \varepsilon\text{-closure}(\delta'(q_0, 1) \cup \delta'(q_1, 1) \cup \delta'(q_2, 1))\\ &= \varepsilon\text{-closure}(\{Ø\} \cup \{q_1\} \cup \{Ø\})\\ &= [q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta_D [q_0, q_1, q_2], 2) &= \varepsilon\text{-closure}(\delta'([q_0, q_1, q_2], 2))\\ &= \varepsilon\text{-closure}(\delta'(q_0, 2) \cup \delta'(q_1, 2) \cup \delta'(q_2, 2))\\ &= \varepsilon\text{-closure}(\{Ø\} \cup \{Ø\} \cup \{q_2\})\\ &= [q_2]\end{aligned}$$

The next new state is [q_1, q_2] and the transitions for the new states are

$$\begin{aligned}\delta_D([q_1, q_2], 0) &= \varepsilon\text{-closure}(\delta'([q_1, q_2], 0))\\ &= \varepsilon\text{-closure}(\delta'(q_1, 0) \cup \delta'(q_2, 0))\\ &= \varepsilon\text{-closure}(\{Ø\} \cup \{Ø\})\\ &= [Ø]\end{aligned}$$

$$\begin{aligned}\delta_D([q_1, q_2], 1) &= \varepsilon\text{-closure}(\delta'([q_1, q_2], 1))\\ &= \varepsilon\text{-closure}(\delta'(q_1, 1) \cup \delta'(q_2, 1))\\ &= \varepsilon\text{-closure}(\{q_1\} \cup \{Ø\})\\ &= [q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta_D([q_1, q_2], 2) &= e\text{-closure}(\delta'([q_1, q_2], 2))\\ &= \varepsilon\text{-closure}(\delta'(q_1, 2) \cup \delta'(q_2, 2))\\ &= \varepsilon\text{-closure}(\{Ø\} \cup \{q_2\})\\ &= [q_2]\end{aligned}$$

The third new state is [q_2].

$$\begin{aligned}\delta_D([q_2], 0) &= \varepsilon\text{-closure}(\delta'([q_2], 0))\\ &= \varepsilon\text{-closure}(\delta'(q_2\ 0))\\ &= \varepsilon\text{-closure}(\{\emptyset\})\\ &= [\emptyset]\end{aligned}$$

$$\begin{aligned}\delta_D[q_2], 1) &= \varepsilon\text{-closure}(\delta'([q_2], 1))\\ &= \varepsilon\text{-closure}(\delta(q_2\ 1))\\ &= \varepsilon\text{-closure}(\{\emptyset\})\\ &= [\emptyset]\end{aligned}$$

$$\begin{aligned}\delta_D([q_2], 2) &= \varepsilon\text{-closure}(\delta'([q_2], 2))\\ &= \varepsilon\text{-closure}(\delta'(q_2\ 2))\\ &= \varepsilon\text{-closure}(\{q_2\}) = [q_2]\end{aligned}$$

The last new state is [Ø] which is a dummy state, and the transitions are

$$\begin{aligned}\delta_D[\emptyset], 0) &= \varepsilon\text{-closure}(\delta'([\emptyset], 0))\\ &= \varepsilon\text{-closure}(\{\emptyset\})\\ &= [\emptyset]\end{aligned}$$

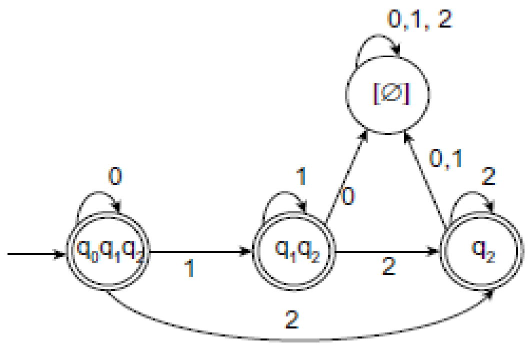
$$\begin{aligned}\delta_D([\emptyset], 1) &= \varepsilon\text{-closure}(\delta'([\emptyset], 1))\\ &= \varepsilon\text{-closure}(\{\emptyset\})\\ &= [\emptyset]\end{aligned}$$

$$\begin{aligned}\delta_D([\emptyset], 2) &= \varepsilon\text{-closure}(\delta'([\emptyset], 2))\\ &= \varepsilon\text{-closure}(\{\emptyset\})\\ &= [\emptyset]\end{aligned}$$

DFA transition table

	a = 0	a = 1	a = 2
→ $[q_0 q_1 q_2]$	$[q_0 q_1 q_2]$	$[q_1 q_2]$	$[q_2]$
*$[q_1 q_2]$	[Ø]	$[q_1 q_2]$	$[q_1 q_2]$
*$[q_2]$	[Ø]	[Ø]	$[q_2]$
[Ø]	[Ø]	[Ø]	[Ø]

The resultant DFA for the given NFA-ε is



1.3.4. Simulation of NFA:

Input: an input string **x** terminated by **eof** character. An NFA N with start state S_0, Accepting states F and transition function move.

Output: Answer either "Yes" or "No".

Method: The algorithm keeps a set of current states S, that are reachable from s_0. If c is a next input character, read by function nextChar(), then we first compute move(S,c) and then compute ε-closure( ).

Algorithm:

```
S=epsilon-closure(s0)
c=nextChar()
while(c!=eof)
{
S=e-closure(move(S,c);
c=nextChar();
}
if(S ⋂ F!= ∅ )
return "yes";
else
return "no";
```

1.3.5. Construction of an NFA-ε from regular expression:

Algorithm: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA.

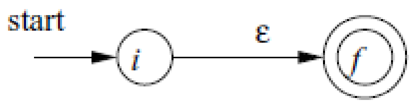
Input: A regular expression **r** over **Σ**.

Output: An NFA N accepting **L(r)**.

Method:

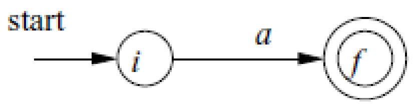
Basis:

i) For expression ε construct NFA.



State **i** is the starting state and **f** is the final state.

ii) For any sub-expression a in Σ, construct NFA.



Induction:

Suppose N(s) and N(t) are NFA's for regular expression s and t, respectively.

a) Suppose **r=s | t**. Then N(r), the NFA for r, is constructed as shown in figure below.

Here i and f are the new states, the start and accepting states of N(r).

Note: The final states of N(s) and N(t) are not accepting states in N(r).

N(r) accepts L(s) U L(t).

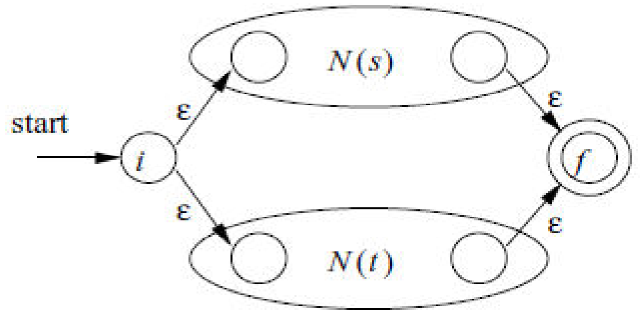


Fig: NFA for union of two regular expressions

b) Suppose r=st. Then N(r), the NFA for r, is constructed as shown in figure below.
The start state of N(s) becomes the start state of N(r), and the final state of N(t) is only the accepting state of N(r). The accepting state of N(s) and N(t) are merged into a single state, with all the transitions in or out of either state. N(r) accepts L(s)L(t).

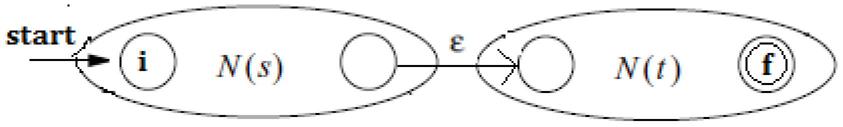


Fig: NFA for the concatenation of two regular expressions

c) Suppose r=s*. Then N(r), the NFA for r, is constructed as shown in figure below.
Here i and f are the new states, the start and accepting states of N(r). N(r) accepts L(r)*.

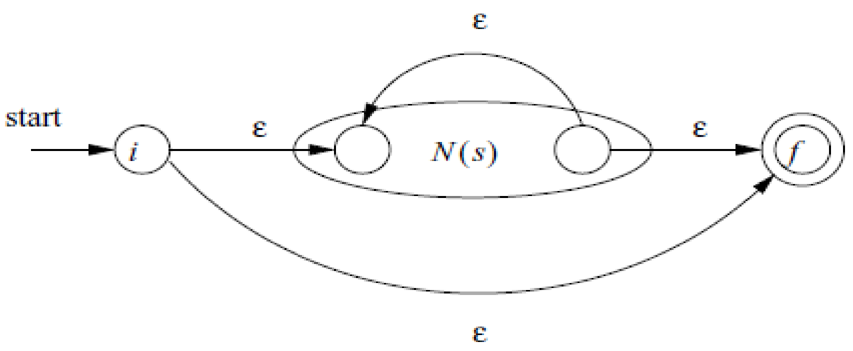


Fig: NFA for closure of regular expressions

d) Suppose r=(s). Then L(r) =L(s) and we can use the NFA N(s) as N(r).

Problem: Construct NFA for the regular expression **(a|b)*abb.**
Solution: Divide the given regular expression into sub expressions to construct the NFA. The sub expressions can be interpreted by using the following parse tree.

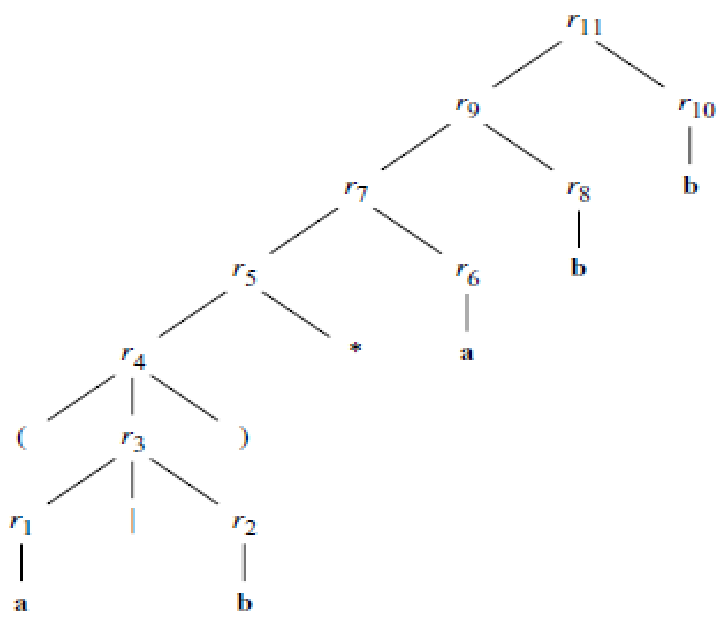
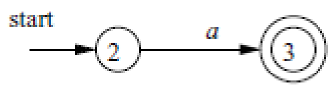
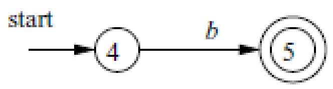


Fig.: Parse Tree for the regular expression (a|b)*abb

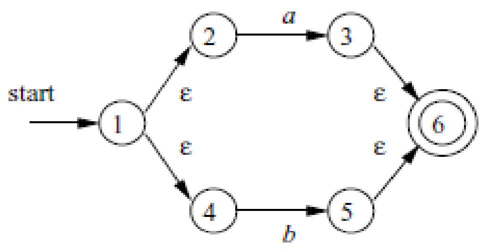
Now r1=a. N(r1) will be constructed by applying basis rule (ii) as follows:



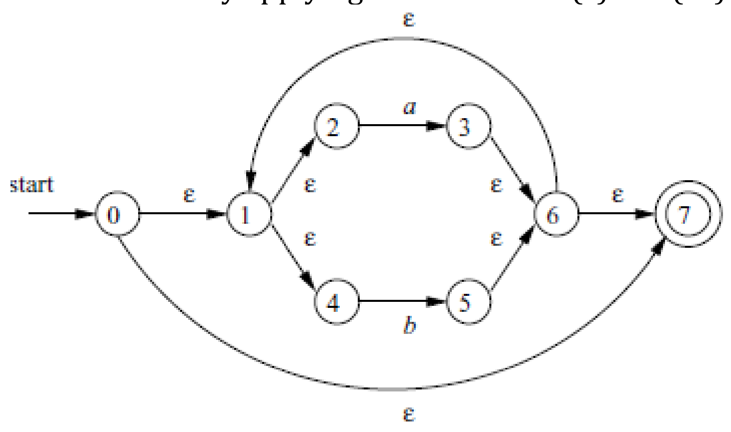
r2=b, N(r2) will be constructed by applying basis rule (ii) as follows:



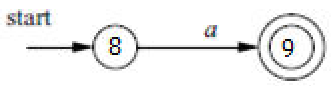
Now r3=r1 | r2, N(r3) can be obtained by applying induction rule (a).



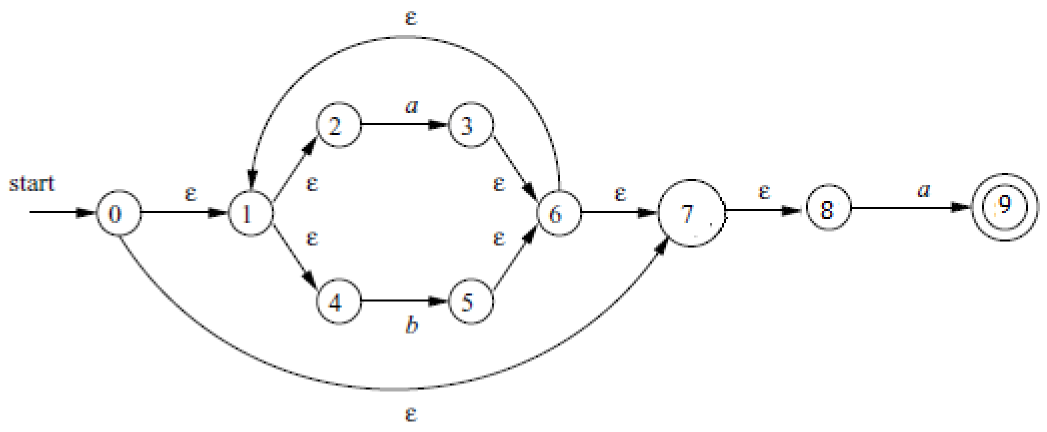
Then r4=(r3). N(r4) is same as N(r3) as per induction rule (d).
NFA for r5 can be constructed by applying induction rule (c). r5=(r4)*



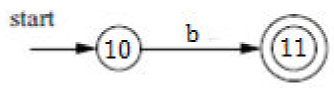
NFA for r6 can be constructed as follows.



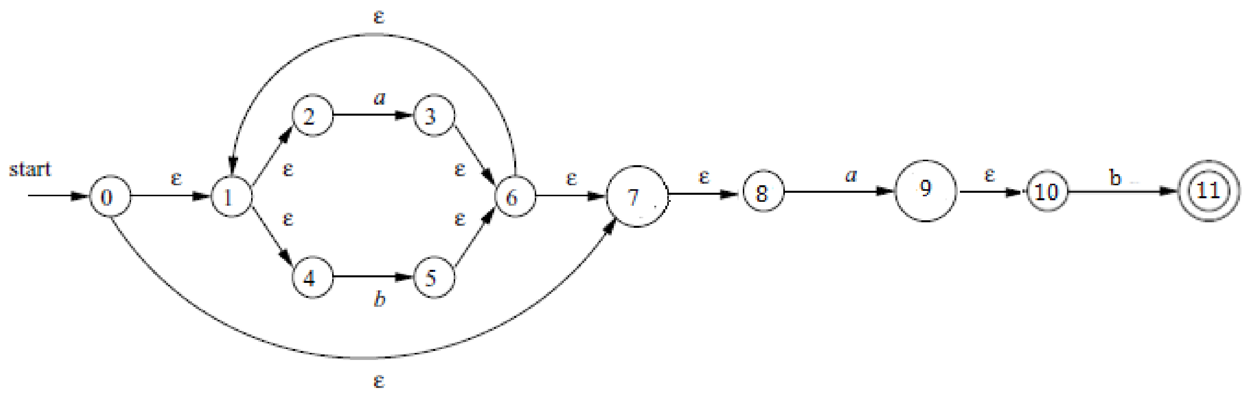
To obtain NFA for r7=r5.r6 by applying induction rule (b).



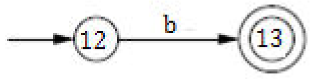
NFA for r8 can be constructed as follows



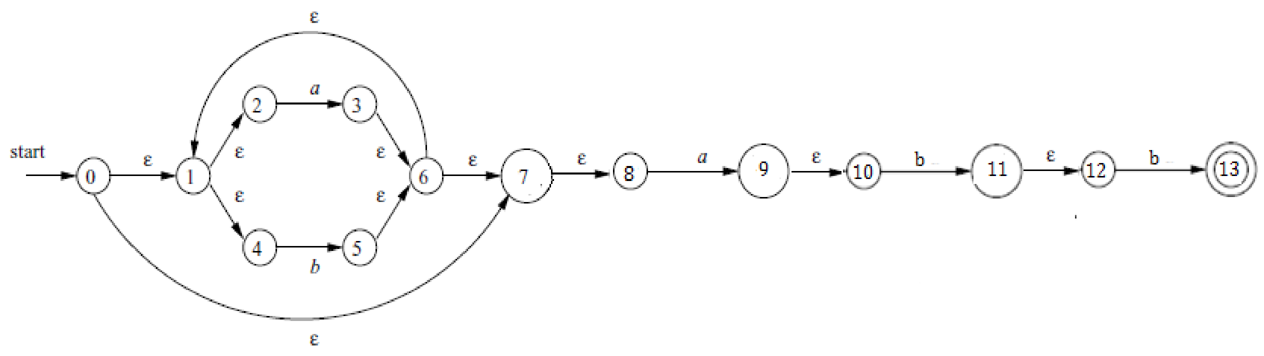
NFA for r9 can be constructed by applying induction rule (b).
r9=r7.r8



NFA for r10 can be constructed as follows.



NFA for r11 can be constructed by applying induction rule (b).
r11=r9.r10
NFA(r11) will give the final NFA for the given regular expression **(a|b)*abb.**



1.3.6. Design of Lexical Analyzer Generator

i) The Structure of Lexical Analyzer Generator

A LEX program simulates an automaton. Usually, a Lex program is transformed into a transition table and actions which are used by finite automaton simulator.

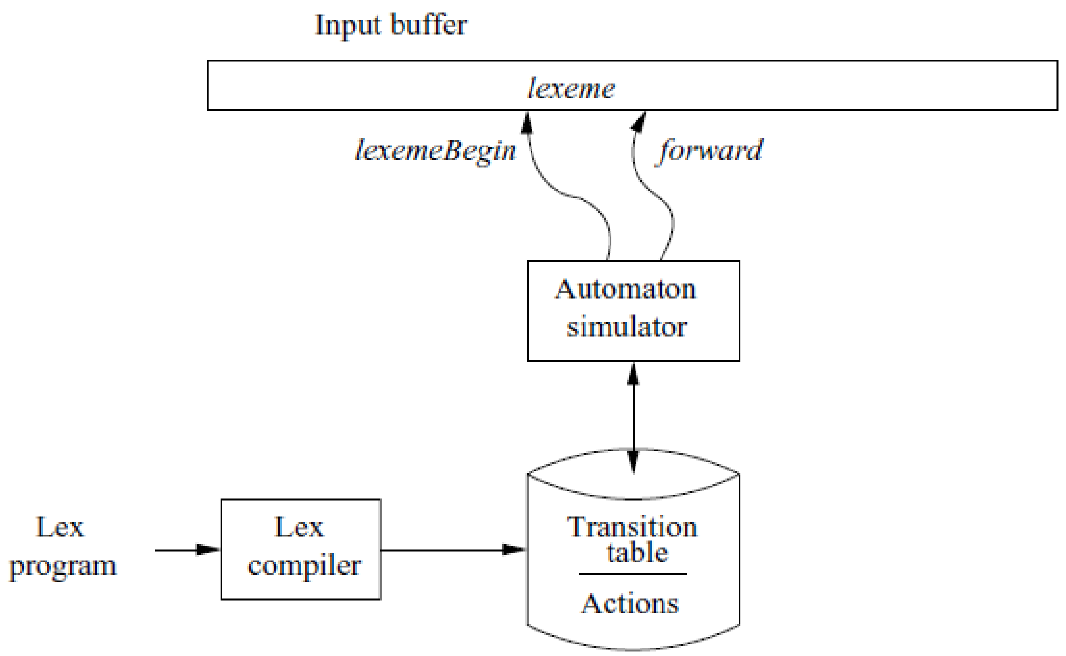


Fig.: Structure of Lexical Analyzer Generator (LEX)

The following components are created by a lex program.

a) Transition Table.

b) The functions that are passed directly through LEX to the output.

c) The actions from the input program which appear as a fragment of code to be invoked at appropriate time by the automaton simulator.

To construct automaton, the regular expressions specified in the lex program for recognizing various patterns/lexemes are converted into a single NFA by using algorithm explained in previous section.

ii) Pattern Matching Based on NFA's

To simulate NFA, the lexical analyzer has to read the input by using **lexemebegin** pointer**. As** it moves the **forward pointer** ahead in the input, it determines the set of states it is in each point. Finally, the NFA simulation reaches a point on the input where there are no next states. At this point, we can decide the longest prefix (lexeme) matching some pattern. We look backward in the sequence of set of states, until we find a set that includes one or more accepting states. If there are several accepting states in that set, we consider the one associated with earliest patter **P_i** in the list from the lex program. Move the forward pointer back to the end of the lexeme and perform the action **A_i** associated with pattern P_i

Example: NFA for **a,ab** and **a^*b^+**

Sample lex program to recognize above patterns

a { action A_1 for pattern P_1}
ab { action A_2 for pattern P_2}
a^*b^+ { action A_3 for pattern P_3}

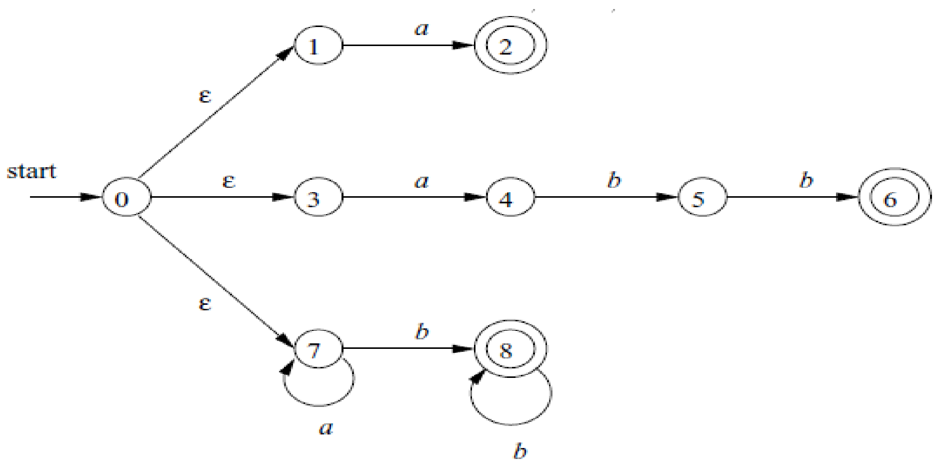


Fig. *NFA to recognize* ***a,ab and a^*b^+***

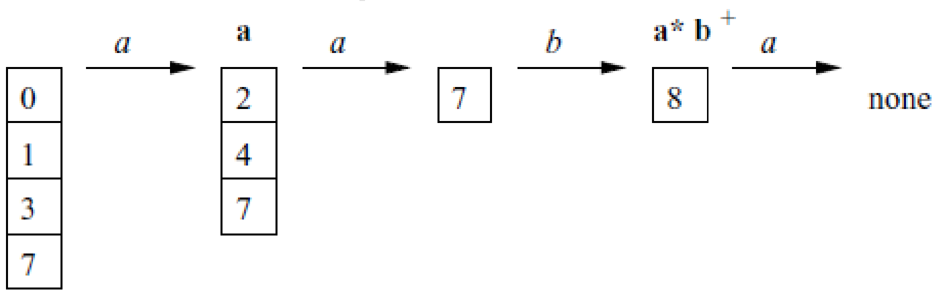


Fig. *Sequence of states obtained on processing input* ***aaba***

The above figure shows the set of states of NFA that we enter while processing aaba. We start with ε-closure of state 0, so we get state set {0,1,3,7}. After reading the forth symbol, we are in empty set of states, since there are no transitions from state 8 on input symbol a. Thus, we need to backup to determine the set of states that includes an accepting state. After reading aab we are in state 8, which indicates that **a^*b^+** has been matched: aab is the logest prefix that gets to an accepting state. So we select lexeme as aab and execute its action which should return to the parser indicating the token whose pattern is P_3 **$=a^*b^+$** has been found.

iii) DFA's for Lexical Analyzer

Another architecture resembling the output of Lex is to convert the NFA for all the patterns into an equivalent DFA using subset construction algorithm.

1.3.7. Optimization of DFA-Based Pattern Matchers

Lex compiler constructs DFA directly from regular expression without constructing an intermediate NFA.

Converting a regular expression directly to a DFA.

Input: A regular expression r.

Output: A DFA D that recognizes L(r).

Method:

i) Construct a syntax tree T from the augmented regular expression (r)#.

ii) Compute nullable, firstpos, lastpos, and followpos for T.

iii) Construct Dstates, the set of states of DFA D, and Dtran, the transition function for D.

The states of D are sets of positions in T. Initially, each state is “unmarked," and a state becomes “marked" just before we consider its out-transitions. The start state of D is firstpos(n_0), where node n_0 is the root of T. The accepting states are those containing the position for the end marker symbol #.

(OR)

Algorithm to convert a regular expression to DFA directly.

i) Formulate augmented regular expression from the given regular expression. The augmented regular expression is **(a|b)*abb#.**

ii) Construct syntax tree for the augmented regular expression.
In syntax tree, leaves correspond to the operands and interior nodes corresponds to operators. An interior node is called a **cat-node**, **or-node** or **star-node** if it is labelled by a concatenation (dot), union operator |, or star operator * respectively.

iii) Assign a number to each leaf node in a syntax tree.

iv) Compute **nullable** of each node.
nullable(n) is true for a syntax-tree node n if and only if the sub-expression represented by n has ε in its language.

v) Compute **firstpos** of each node.
firstpos(n) is the set of positions in the subtree rooted at n that correspond to the first symbol of at least one string in the language of the sub-expression rooted at n.

vi) Compute **lastpos** of each node.
lastpos(n) is the set of positions in the subtree rooted at n that correspond to the last symbol of at least one string in the language of the sub-expression rooted at n.
Rules for computing nullable, firstpos and lastpos is shown in table below.

Rules for computing nullable, firstpos and lastpos

Node n	nullable(n)	firstpos(n)	lastpos(n)
A leaf node labelled ε	True	∅	∅
A leaf node labelled i	False	{i}	{i}
A star-node **n=c1***	True	firstpos(c1)	lastpost(c2)

A or-node **n=c1 | c2**	nullable(c1) or nullable(c2)	firstpos(c1) ∪ firstpost(c2)	lastpost(c1) ∪ lastpost(c2)
A cat-node n=c1.c2	nullable(c1) and nullable(c2)	if nullable(c1) firstpos(c1) ∪ firstpost(c2) else firstpost(c2)	if nullable(c2) lastpost (c1) ∪ lastpost (c2) else lastpost (c2)

vii) Compute **followpos** for cat-node and star-node in a syntax tree.

Rules for computing followpos

a) If n is a cat-node with left child c1 and right child c2, then for every position i in lastpos(c1), all positions in firstpos(c2) are in followpos(i).
firstpos(c2) →lastpos(c1)

b) If n is a star-node, and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i). *firstpos(n)→lastpos(n)*

viii) Consider **firstpos(root)** as start state of DFA and unmark it.
Compute the transitions from start state on every input symbol. If any new state is obtained add it to the state set by unmarking it. Repeat this process until all the states in state set are marked. Construct the DFA from the transitions obtained.

Example: Construct DFA for the regular expression **(a|b)*abb**

Solution:

i) Formulate augmented regular expression as (a|b)*ab#.

ii) Construct syntax tree for augmented regular expression.

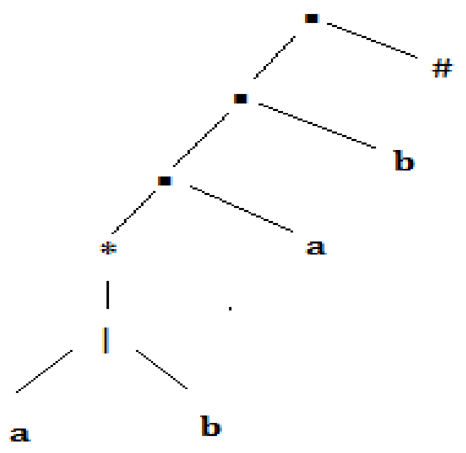


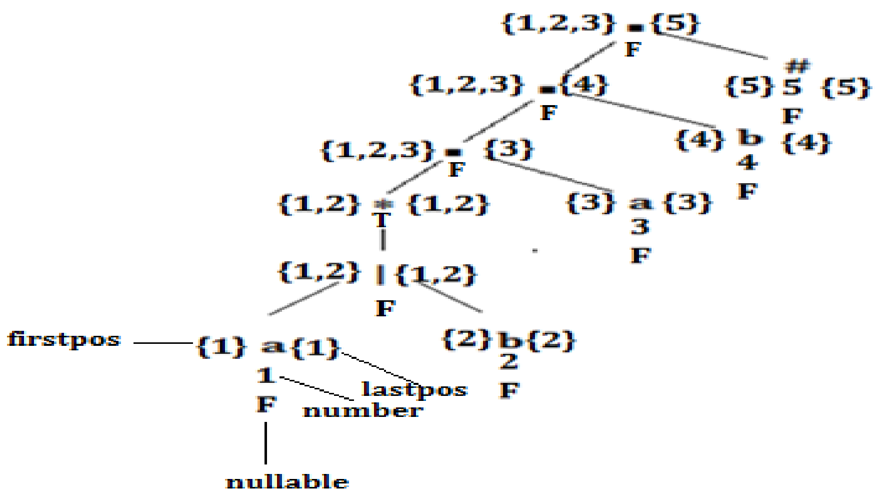
Fig: Syntax Tree

iii) Assign number to each leaf node.

iv) Compute nullable of each node.

v) Compute firstpos of each node

vi) Compute followpos of each node.

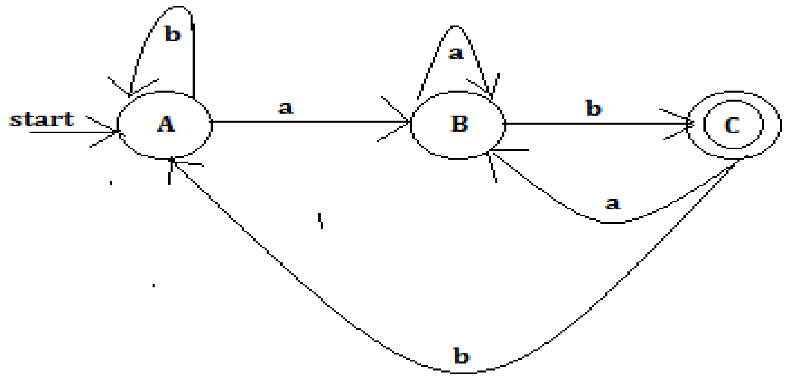


vii) Compute **followpos**.

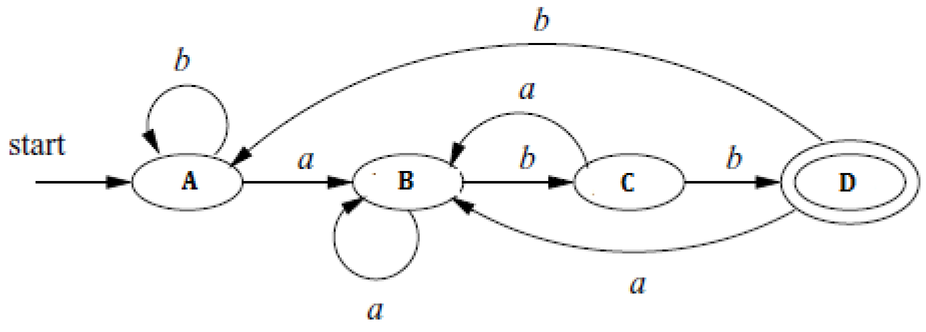
nodes	*followpos*
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	∅

viii) Consider root node of syntax tree & compute its firstpos. The result of firstpos of root node is considered as start state of DFA.
firstpost(root)={1,2,3}
Consider {1,2,3} as state A and unmark it.
Now, compute transitions from A on input symbol a.
(A,a)={1,3}
=followpos(1) ∪ followpos(3)
={1,2,3} ∪ {4}
={1,2,3,4}, consider this set as state B and unmark it.
Now, compute transitions from A on input symbol b.
(A,b)={2}
=followpos(2)
={1,2,3}, this is same as state A.
Now consider unmarked state B and compute transitions on input symbol a & b.
(B,a)={1,3}
=followpos(1) ∪ followpos(3)
={1,2,3,4}, this is same as state B.

(B,b)={2,4}
=followpos(2) ∪ followpos(4)
={1,2,3} ∪ {5}
={1,2,3,5} this is a new state, so consider it as state C and unmark it.
Now consider unmarked state C and compute transitions on input symbol a & b.
(C,a)={1,3}
=followpos(1) ∪ followpos(3)
={1,2,3,4}, this is same as state B.
(C,b)={2}
=followpos(2)
={1,2,3}
Since no new states are generated, the process of computing transitions is terminated.
Now, construct the optimized DFA from above states and transitions.
Mark state C as final state because it contains end marker symbol (#).



Example 2: Construct DFA for the regular expression **(a|b)*abb**
Solution:


b
b
a
start
A
a
B
b
C
b
D
a
a