COMPILER DESIGN

UNIT – I

Introduction Language Processing, Structure of a compiler, the Evaluation of Programming language, The Science of building a Compiler application of Compiler Technology. Programming Language Basics.

Lexical Analysis-:The role of lexical analysis buffering, specification of tokens. Recognitions of tokens the lexical analyzer generator lexical

UNIT -1

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the HLL program input into an equivalent machine language program.
2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers. Important role of translator are:

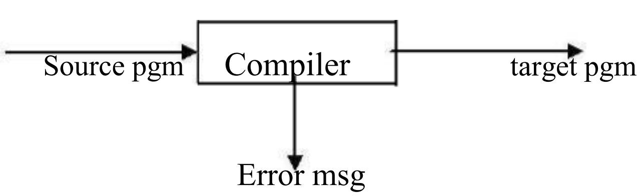
1 Translating the hll program input into an equivalent ml program.
2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

a. **Compiler**
b. **Interpreter**
c. **Preprocessor**

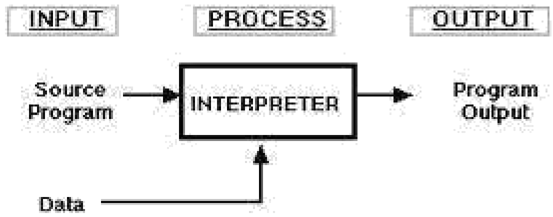
Compiler

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Executing a program written n HLL programming language is basically of two parts. The source program must first be compiled and translated into a object program. Then the resulting object program is loaded into a memory executed.

Interpreter*:* An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.
1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.

5. Type of object that denotes a various may change dynamically.

Debugging a program and finding errors is simplified task for a program used for interpretation.

The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is *slower*.

Memory consumption is more.

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

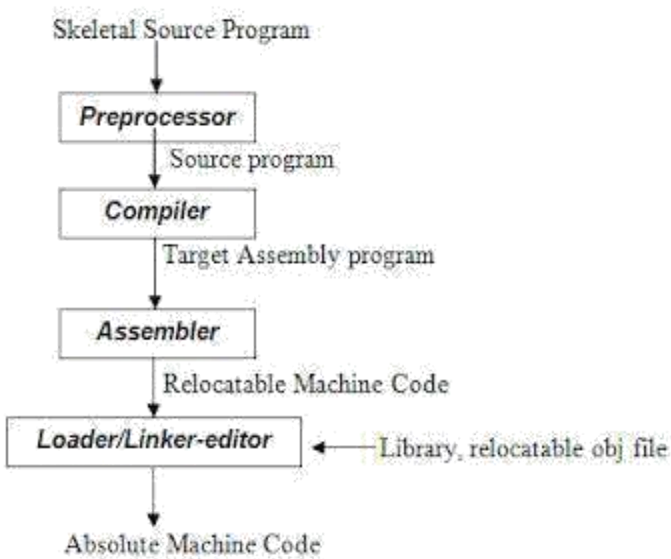


Fig 1.1 Language –processing System

Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

Assembler*:* programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user"s program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could"relocate" directly behind the user"s program. The task of adjusting programs othey may be placed in arbitrary core locations is called relocation.

STRUCTURE OF A COMPILER

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called p**hases'**.

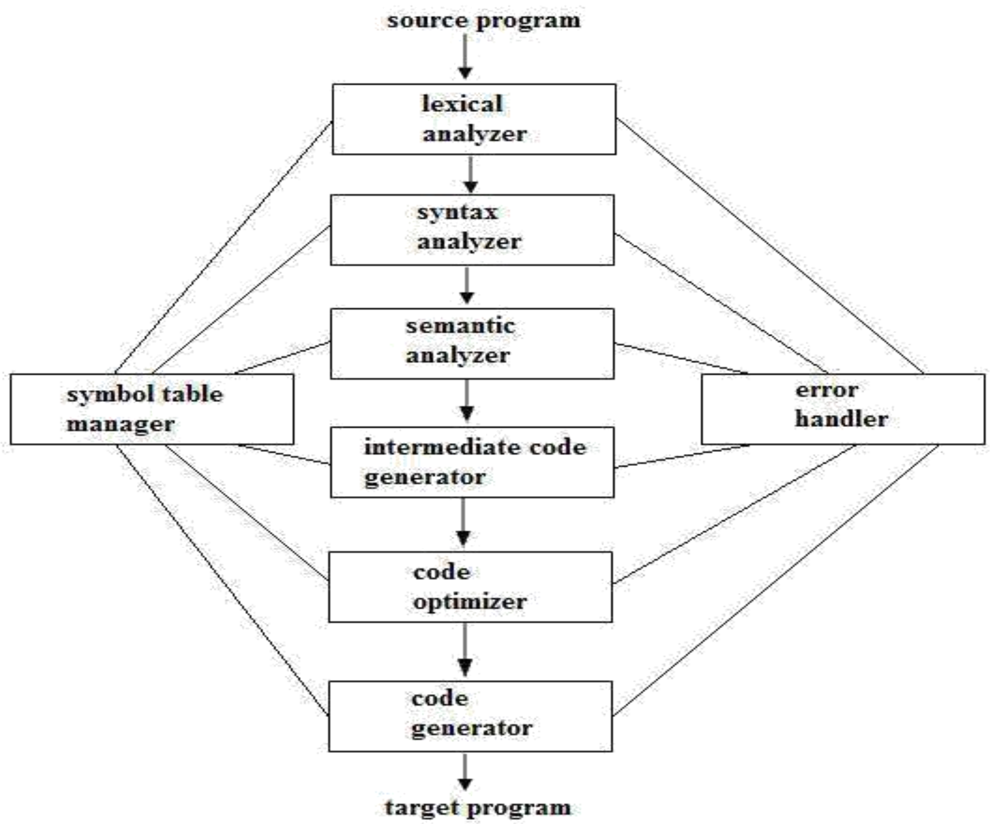


Fig 1.5 Phases of a compiler

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a „Symbol Table".

Error Handlers:-

It is invoked when a flaw error in the source program is detected.

The output of **LA** is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)

1, divide A by B and then multiply by C or

2, multiply B by C and then use the result to divide A.

each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands.

The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

1, Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**

Goto **L3**

L2 :

This can be replaced by a single statement

If **A < B** goto **L3**

Another important local optimization is the elimination of common sub-expressions

A:=B+C+D
E:=B+C+F

Might be evaluated as

T1 :=B+ C

A :=T1+ D
E :=T1+ F

Take this advantage of the common sub-expressions **B + C.**

2, Loop Optimization:-

Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

Code Generator :-

Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book-keeping :-

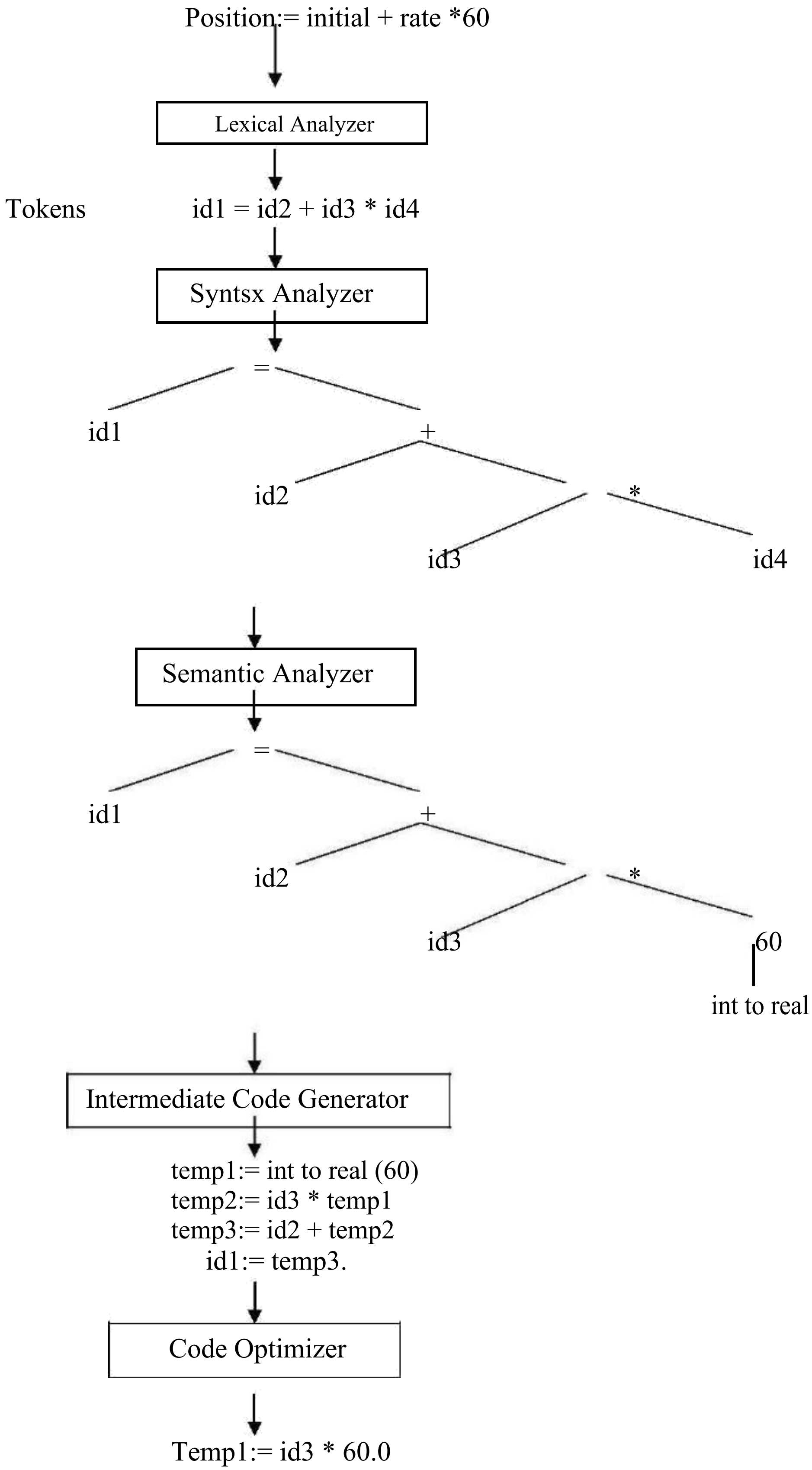
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Error Handing :-

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:


Position:= initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntsx Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real
Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer
Temp1:= id3 * 60.0

Id1:= id2 +temp1

↓

Code Generator

↓

```
MOVF id3, r2
MULF   *60.0, r2
MOVF   id2, r2
ADDF    r2, r1
MOVF   r1, id1
```

Evolution of Programming languages

The **history of programming languages** spans from documentation of early mechanical computers to modern tools for software development. Early programming languages were highly specialized, relying on mathematical notation

The move to Higher Level Languages

The first step towards more people friendly programming languages was the development of mnemonic assembly languages in the early 1950's.The instructions in assembly languages were just mnemonic representations of machine instructions.

A major step towards higher level languages was made in the later half of the 1950's with the development of FORTRAN for scientific computation, Cobol for business data Processing and Lisp for symbolic computation.

In the following decades many more languages were created with innovative features to help make programming easier, more natural, and more robust.

Languages can also be classified in variety of ways.

Classification by Generation: Ist generation are the machine languages, 2nd generation are the assembly languages, 3rd generation are the higher level languages like Fortran,cobol, Lisp,C etc.4th generation are the languages designed for specific application like NOMAD,SQL,POST The term fifth generation language has been applied to logic and the constraint based language like prolog and OPS5.

Classification by the use: imperative languages in which your program specifies How computation is to be done the declarative for languages in which your program specifies what computation is to be done.

Examples:

Imperative languages: C,C++,C#,Java.

Declarative languages: ML, Haskell, Prolog

Object oriented language is one that supports Object oriented programming, a Programming style in which a program consists of a collection of objects that interact with one another.

Examples: Simula 67, small talk, C ++, Java,Ruby

Scripting languages are interpreted languages with high level operators designed for "gluing together" computations These computations originally called Scripts

Example: JavaScript, Perl, PHP, python, Ruby, TCL

The Science of building a Compiler

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. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modelling in compiler design and implementation: The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms.Some of most fundamental models are finite-state machines and regular expressions.These models are useful for de-scribing 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 such as the nesting of parentheses or control constructs. Similarly, trees are an important model for representing the structure of programs and their translation into object code.

The science of code optimization: The term "optimization" in compiler design refers to the attempts that a com-piler makes to produce code that is more efficient than the obvious code. In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively par-allel computers require substantial optimization, or their performance suffers by orders of magnitude.

Compiler optimizations must meet the following design objectives:

1. The optimization must be correct, that is, preserve the meaning of the compiled program,
2. The optimization must improve the performance of many programs,
3. The compilation time must be kept reasonable, and
4. The engineering effort required must be manageable.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems.

Applications of Compiler Technology

Compiler design impacts several other areas of computer science.

Implementation of high-level programming language: A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler

must translate that program to the target language. higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code.
Language features that have stimulated significant advances in compiler technology.
Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations,* has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C + + , C # , and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

Java has many features that make programming easier, many of which have been introduced previously in other languages. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.
In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.
Optimizations for Computer Architecture: high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies.* Parallelism can be found at several levels: at the *instruction level,* where multiple operations are executed simultaneously and at the *processor level,* where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.
Design of New Computer Architectures: in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features. One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture.
Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize. Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept.

Specialized Architectures Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Program Translations: The following are some of the important applications of program-translation techniques.

Binary Translation: Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines.

Hardware Synthesis: Not only is most software written in high-level languages; even hardware de-signs are mostly described in high-level hardware description languages like Verilog and VHDL. Hardware designs are typically described at the register trans-fer level (RTL), where variables represent registers and expressions represent combinational logic.

Database Query Interpreters: Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or com-piled into commands to search a database for records satisfying that predicate.

Programming Language Basics:

1 The Static/Dynamic Distinction

2 Environments and States

3 Static Scope and Block Structure

4 Explicit Access Control

5 Dynamic Scope

6 Parameter Passing Mechanisms

The Static/Dynamic Distinction: Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time.* On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy.* One issue is the scope of declarations. 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.*

Environments and States:

The *environment* is a mapping from names to locations in the store. Since variables refer to locations, 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 1-values to their corresponding r-values, in the terminology of C. Environments change according to the scope rules of a language.

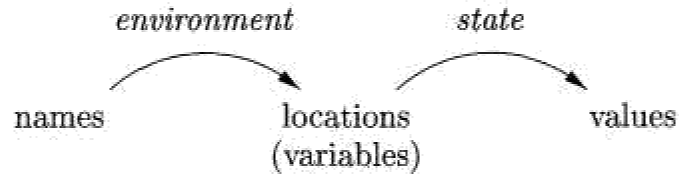


Figure 1.8: Two-stage mapping from names to values

Static Scope and Block Structure

Most languages, including C and its family, use static scope. we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

A C program consists of a sequence of top-level declarations of variables and functions.Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x.*

A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces. 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 a block-structured lan-guages 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 J5, in which x is redeclared. Here, x is redeclared in B' if some other declaration D' of the same name x belongs to B'.

An equivalent way to express this rule is to focus on a use of a name x. Let Bi, i?2, • • • , Bk be all the blocks that surround this use of x, with Bk the smallest, nested within Bk-i, which is nested within Bk-2, and so on. Search for the largest i such that there is a declaration of x belonging to B^. This use of x refers to the declaration in B{. Alternatively, this use of x is within the scope of the declaration in Bi.

Explicit Access Control

Through the use 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 superclass. These keywords support *encapsulation* by restricting access. Thus, 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.

Dynamic Scope

Any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two ex-amples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

Declarations and Definitions

Declarations tell us about the types of things, while definitions tell us about their values. Thus, i n t i is a declaration of i, while i = 1 is a definition of i.

The difference is more significant when we deal with methods or other procedures. In C + + , a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the signature for the method. The method is then defined, i.e., the code for executing the method is given, in another place. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

Parameter Passing Mechanisms

In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both.

Call - by - Value

In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C + + , as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as param eters in C, C + + , or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if a is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter *x,* then an assignment such as x [i] = 2 really changes the array element a[2]. The reason is that, although *x* gets a copy of the value of *a,* that value is really a pointer to the beginning of the area of the store where the array named *a* is located.

Similarly, in Java, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays, strings, and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

Call - by - Reference

In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference is used for "ref" parameters in C + + and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call - by - Name

A third mechanism — call-by-name — was used in the early programming language Algol **60.** It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

main() {

```
main() {
    int a = 1;                                  B1
    int b = 1;
    {
        int b = 2;                          B2
        {
            int a = 3;              B3
            cout << a << b;
        }
        {
            int b = 4;              B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

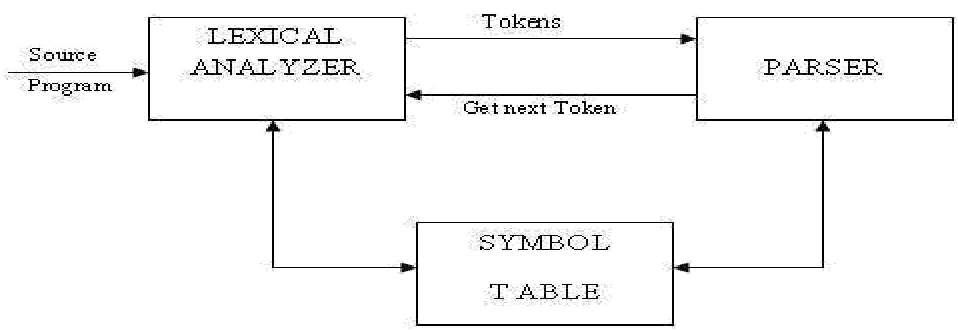
LEXICAL ANALYSIS

2.1 OVER VIEW OF LEXICAL ANALYSIS

- o To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- o Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

2.2 ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a get next token command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	**Parsing**
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

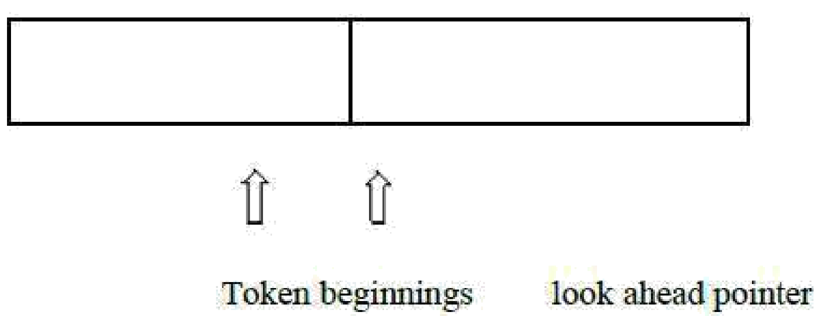
2.3 INPUT BUFFERING

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2… ARG *n*) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

2.4 TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	**lexeme**	**pattern**
const	const	const
if	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

2.5 LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

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

2.6 DIFFERENCE BETWEEN COMPILER AND INTERPRETER

A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.

Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.

List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.

An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

The compiler produce object code whereas interpreter does not produce object code. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.

Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.

Example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

2.7 REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R2R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each „a‟ in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol „a‟ .
- If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

2.8 REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|
Digits - 0|1|2|….|9
letter (letter / digit)* I

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

Stmt -> if expr then stmt
| If expr then else stmt
| є
Expr --> term relop term
|term
Term -->id

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

digit -->[0,9]
digits -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter -->[A-Z,a-z]
id -->letter(letter/digit)*
if --> if
then -->then
else -->else
relop --></>/<=/>=/==/< >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws --> $(blank/tab/newline)^{+}$

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 parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	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	ET
< >	relop	NE

2.9 TRANSITION DIAGRAM:

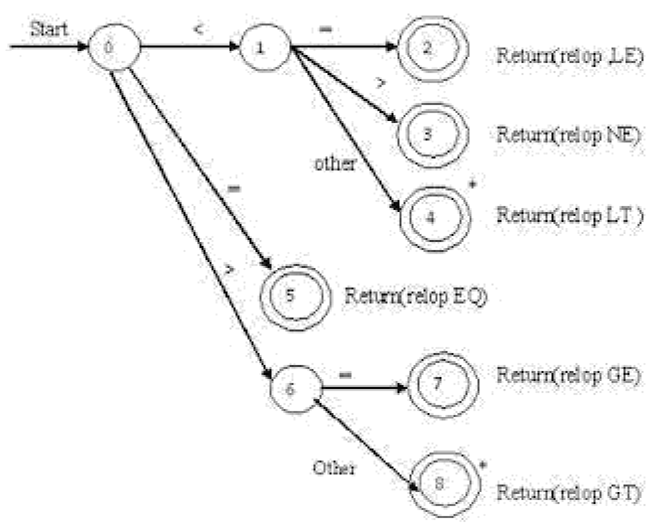
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

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

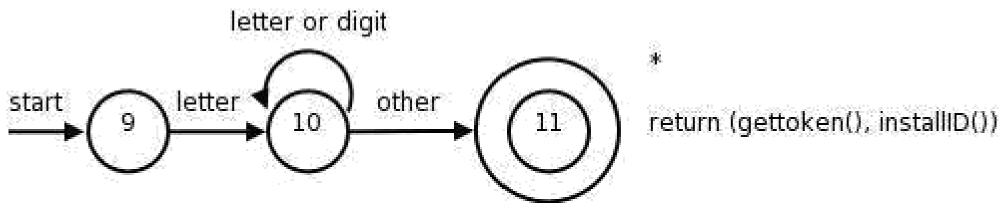
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

If	=	if
Then	=	then
Else	=	else
Relop	=<|<=|=|>|>=	
Id	=letter (letter | digit) *|	
Num	=	digit |

2.10 AUTOMATA

An automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1, an automation in which the output depends only on the input is **called an automation without memory.**

2, an automation in which the output depends on the input and state also is **called as automation with memory**.

3, an automation in which the output depends only on the state of the machine is **called a Moore machine**.

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

2.11 DESCRIPTION OF AUTOMATA

1, an automata has a mechanism to read input from input tape,

2, any language is recognized by some automation, Hence these automation are basically language acceptors or language recognizers.

Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

2.12 DETERMINISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1, it has no transitions on input € ,

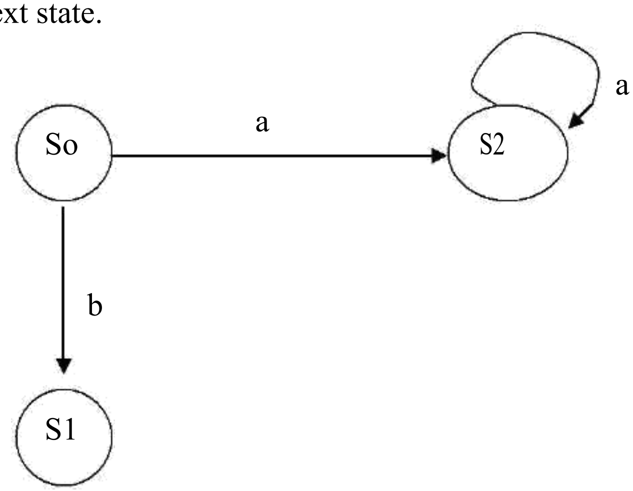
2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where
Q is a finite „set of states", which is non empty.
∑ is „input alphabets", indicates input set.
qo is an „initial state" and qo is in Q ie, qo, ∑, Q
F is a set of „Final states",
δ is a „transmission function" or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input „a" there is only one path going to S2. similarly from S0 there is only one path for input going to S1.

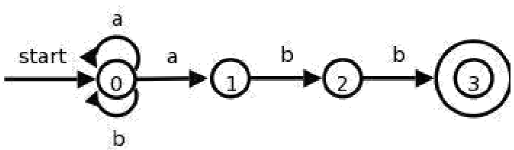
2.13 NONDETERMINISTIC AUTOMATA

- A NFA is a mathematical model that consists of
  - A set of states S.
  - A set of input symbols ∑.
  - A transition for move from one state to an other.
  - A state so that is distinguished as the start (or initial) state.
  - A set of states F distinguished as accepting (or final) state.
  - A number of transition to a single symbol.

- A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function.

- This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as by input symbols.

- The transition graph for an NFA that recognizes the language (a | b) * abb is shown



2.14 DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N-T, start symbol and production.

- Terminal are basic symbols form which string are formed.
- N-terminals are synthetic variables that denote sets of strings
- In a Grammar, one N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.
- The production of the grammar specify the manor in which the terminal and N-T can be combined to form strings.
- Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

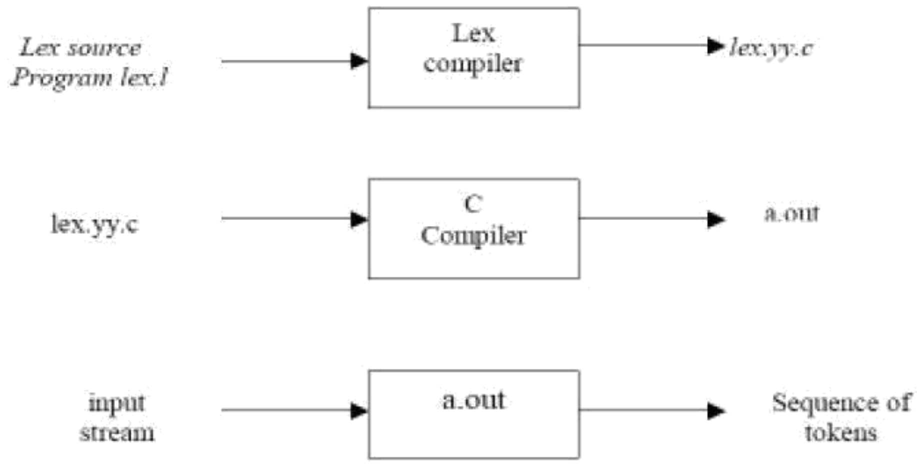
2.15 DEFINITION OF SYMBOL TABLE

- An extensible array of records.
- The identifier and the associated records contains collected information about the identifier.

  FUNCTION identify (Identifier name)

  RETURNING a pointer to identifier information contains
- The actual string
- A macro definition A
- keyword definition
- A list of type, variable & function definition
- A list of structure and union name definition
- A list of structure and union field selected definitions.

2.16 Creating a lexical analyzer with Lex



2.17 Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

   p1 *{action 1}*
   p2 *{action 2}*
   p3 *{action 3}*
   ... ...
   ... ...

   where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.