WINTER - 13 (CT)

Q.1. Explain analysis - synthesis model of compiler in brief. **5M**

Ans. P.1-9, Q.11

Q.2. Explain with suitable examples, the concept of l-value and r-value of an expression **4M**

Ans. P.1-27, Q.39

Q.3. Explain with suitable example, what is boot-strapping **4M**

Ans. P.1-8, Q.10

SUMMER - 14 (CS)

Q.1. Draw and explain general structure of compiler design. **6M**

Ans. P.1-10, Q.12

Q.2. What are compiler writing tools? Explain how to use these tools in detail. **7M**

Ans. P.1-14, Q.17

SUMMER - 14 (CT)

Q.1. What are the various stages in the process of generation of the target code from the source code by the compiler? How these stages are broadly classified? Explain each. **10M**

Ans. P.1-10, Q.12

Q.2. Explain the advantages of analysis synthesis model of compilation. **4M**

Ans. P.1-9, Q.11

WINTER - 14 (CS)

Q.1. Explain the following terms :
(I) Syntax and semantics. (II) Source code and object code. **6M**

Ans. P.1-24, Q.35

Q.2. Explain the output of lexical analysis phase. Give the importance of this output in next phases of compiler. **3M**

Ans. P.1-15, Q.19

Q.3. Write notes on LEX **2M**

Ans. P.1-18, Q.27

Note : Also refer Paper Solutions at the end of the book.

SOLVED QUESTION BANK

[Sequence given as per syllabus]

INTRODUCTION

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code) . The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code).

A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone process most programming is, instead, done using a high-level programming language.

This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. Hence compiler is very essential

A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report programmer mistakes.

INTRODUCTION TO COMPILER

Q.1. What is compiler?

Ans. Compiler :

- A compiler is a program that translates a high-level language program into functionally equivalent low-level language program.
- So, a compiler is basically a translator whose source language is the high-level language and the target language is a low-level language.
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

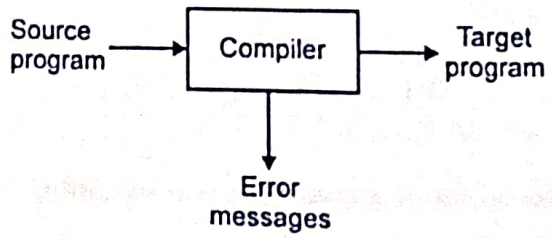


Fig. A compiler

Q.2. What is compilation process? Explain in details.

Ans. Compilation Process :

- Compilation is a process that translates a program in one language (the source language) into an equivalent program in another language (the object or target language).
- Programming language is a problem-oriented language and the target language is a machine language or assembly language (i.e. a machine-oriented language). Thus compilation is a fundamental concept in the production of software.
- It is the link between the (abstract) world of application development and the low-level world of application execution on machines.

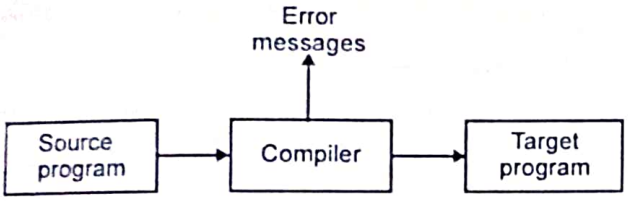


Fig.(a) Compilation process

- Compilation program flowchart can be given as follows :

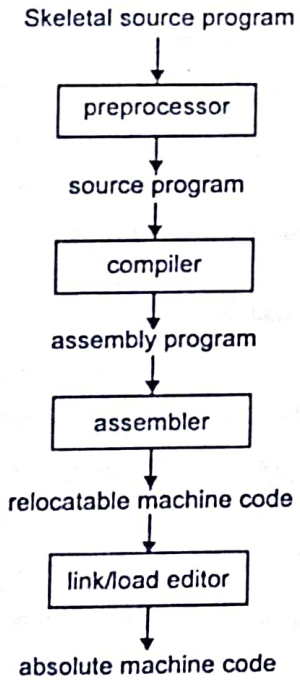


Fig.(b) Compilation program flowchart

Q.3. Define the following :

(i) Tokens (ii) Lexeme

(iii) Patterns

OR Define the following terms :

(i) Phase (ii) Pass

(iii) Pattern (iv) Token

(v) Lexeme

OR What do you mean by phases and phases of compiler?

CS : W-12(4M)

Ans.

(i) Tokens :

- Tokens are symbolic names for the entities that make up the text of the program.
- For example, if for the keyword 'if', and id for any identifier. These make up the output of the lexical analyser.
- Token is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called tokenization.

(ii) Lexeme :

- A lexeme is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token).
- For example, if matches the pattern for if ,and foo123bar matches the pattern for id.

(iii) Patterns :

- A pattern is a rule that specifies when a sequence of characters from the input constitutes a token.
- For example, the sequence if for the token if , and any sequence of alphanumeric starting with a letter for the token id.

(iv) Phase :

- The compiler has to undergo various different individual dependent as well as independent stages of compilation these stages of processing are called as phases of compilation.
- When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases.
- Having separate phases means development can be parceled up into small parts and given to different people.
- It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

(v) **Pass :**

- Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which made a pass over the source (or some representation of it) performing some of the required analysis and translations.
- The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers.
- Many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

COMPILERS AND TRANSLATORS

Q.4. What is translator? **CT : W-10(3M)**

Ans. Translator :

- A translator is a program that takes as input a program written in one programming language and produces output program in another language.
- If source language in high level language such as FORTRAN, PL/I, COBOL and object language is a low level such as assembly language or machine language such type of translator is called as compiler.
- Executing a program written is a high level programming language is a two step process.

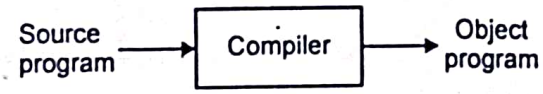


Fig. (1) Compilation

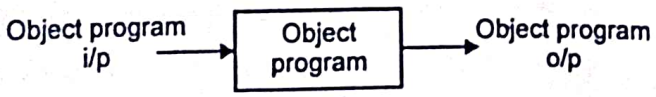


Fig. (2) Execution

- The source program must first be compiled, that is translated into the object program. Then the resulting object program is loaded into memory and executed.

Q.5. State the different types of translators.

Ans. The different types of translators are as follows :

(i) **Interpreter :**

- It translate programming language into simplified langauge called as intermediate code.
- Interpreters are often smaller than compiler and facilitate the implementation of complex programming language constructs.
- The main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

(ii) **Assembler :**

- Programmers find difficulty in writing machine language programs. Hence mnemonics (symbol) are developed for each instruction.
- The mnemonics would easily get converted to machine language programs (codes)
- Such language that is understood by processor using symbols is called as assembly language.
- Programs written to translate the assembly language to machine language is called as assemblers.
- The input to an assembler program is called source program, the output is a machine code called as object program.

(iii) **Preprocessor :** It takes program in one high level language and translate it into the equivalent another high level language.

Q.6. Why we need translators?

Ans.

- With the machine language we must communicate directly with computer in terms of bits, registers and primitive machine operations.
- Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is difficult.
- The main disadvantage of machine language coding is that all operations and operands must be specified in a numeric codes.
- So, we need a translators to modify the machine language program into human readable form.

Q.7. What is the difference between a compiler language and Interpreted language. Give examples of each. When would we prefer interpreter over compiler? **CT : W-11(7M)**

Ans.

Sr. No.	Compiler language	Interpreted language
(1)	In compiler language, the program is first converted into machine code and is then directly executed by CPU.	In this the program is first converted into some interpreted code and is then indirectly executed by interpreter program.
(2)	Reads the whole program called source code written in some high level language and translates code of a programming language in machine code also called as object code.	The interpreter takes one statement then translates it and executes and then takes another statement.
(3)	Compiler language program execution is fast since direct machine code is executed.	Interpreted language program execution is slower, because of indirect code execution.
(4)	The compiler translates the entire program and then executes it.	Interpreter stops translating after the first error.
(5)	Code produced is not portable since it produces machine code which is dependent of machine.	Interpreted code is portable since interpreter language program do not produce machine language program.
(6)	Example of source language are C, Pascal, Python, FORTRAN, LISP etc.	Scripting languages like Java script, ASP, JAVA, etc.
(7)	These are inferior compared to interpreted language with reference to program development environment.	These languages are superior to compiler language as it involves testing, debugging and editing a program which supports program development environment.
(8)	It is costlier as compared to interpreted program development environment.	Interpretation of program is cheaper as compared to compiler language.

CROSS COMPILER

Q.8. What is cross compiler? **CS : S-10,W-13(2M)**

OR Define cross compiler. **CS : S-13(2M)**

OR Write short note on cross compiler. **CT : W-12(3M)**

OR What is the theory and logic behind cross compilation? **CT : W-10(4M)**

Ans. Cross compiler :

- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
- A cross compiler is necessary to compile for multiple platforms from one machine.
- A cross compiler could generate an executable for each of them from one main source.
- Basically there exists three types of languages :

(i) Source language i.e. the application program.

(ii) Target language in which machine code is written.

(iii) The implementation language in which a compiler is written.

- There may be a case that all these three languages are different.
- In other words there may be a compiler which run on one machine and produces the target code for another machine. Such a compiler is called cross compiler. Thus by using cross compiler techniques platform independency can be achieved.
- For example, we have new language L and want to write cross compiler then S to generate code for machine N ; i.e. we create L_SN . If an existing compiler for S runs on machine M and generate code for M, it is characterized by S_MM .
- If L_SN run through S_MM , we get compiler L_MN i.e. compiler from L to N that runs on M.
- This process is illustrated in following Fig. by putting together T diagrams for these compilers.

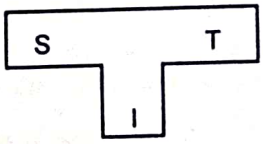


Fig.(a) T diagram with S as source, T as target and I as implementation language

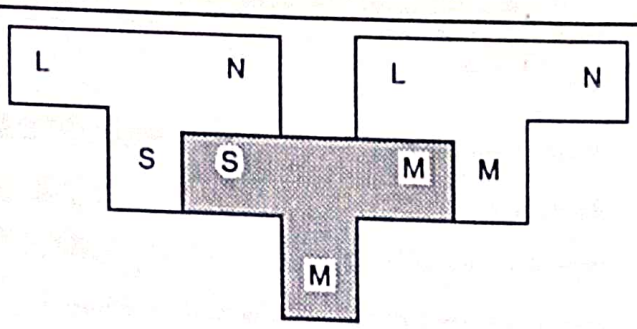


Fig.(b) Cross compiler

- The use of cross compiler makes it possible to overcome the lack of resources by creating an interrelated execution between various components on the system.

Q.9. What is cross assembler? What is the use of cross assembler?

CT : S-11(3M)

OR What is cross assembler? Explain the application of it.

CS : W-11(4M)

OR Explain cross assembler. **CS : S-12(2M)**

Ans. Cross assembler :

- An assembler is a program which creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents.
- An assembler converts assembly language (human readable text) into actual binary processor specific machine code (non human readable binary code).
- An assembler that generates machine language for a different type of computer than the one the assembler is running in.
- It is used to develop programs for computers on a chip or microprocessor used in specialized applications that are either too small or are otherwise incapable of handling the development software.

Use of cross assembler :

(1) Cross assembler are generally used to develop programs which are supposed to run on game consoles, applicances and other specialized small electronics system which are not able to run a development environment.

(2) They can also be used to speed up development for low powered system, for example XASm enables development on a PC based system for a Z80 powered MSX computer.

BOOTSTRAPPING

Q.10. Define bootstrapping. How bootstrapping can be achieved?

CT : W-09(4M), CS : S-13(2M)

OR Explain how bootstrapping is used in design of a compiler.

CS : S-10(3M), W-10(4M), W-12(6M), S-12(2M)

OR What is bootstrapping? **CS : W-13(2M)**

OR Explain with suitable example, what is bootstrapping.

CT : W-13(4M)

Ans. Bootstrapping of compiler :

- Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle far more complicated program.
- This complicated program can further handle even more complicated program and so on.
- Writing a compiler for any high level language is a complicated process.
- It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.
- To clearly understand the bootstrapping technique consider a following scenario :
- Suppose we want to write a cross compiler for new language X.
- The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create X_YZ.
- Now, if existing compiler Y [i.e. compiler written in language Y] runs on machine M and generates code for M then it is denoted as Y_MM. Now if we run X_YZ using Y_MM then we get a compiler X_MZ.
- That means a compiler for source language X that generates a target code in language Z and which runs on machine M.
- Following diagram illustrates the above scenario :

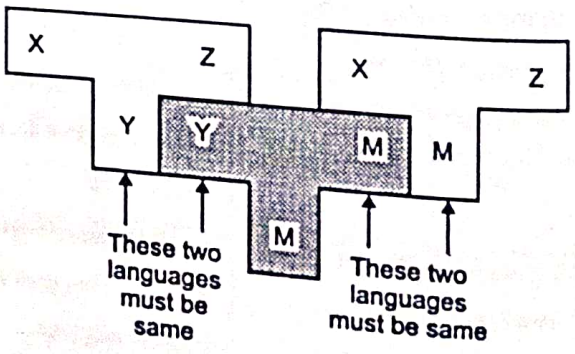
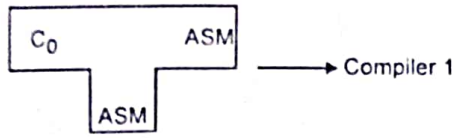


Fig. Bootstrapping process

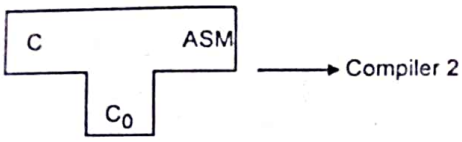
Example :

- We can create compilers of many different forms. Now we will generate compiler which takes C language and generates as assembly language as an output with the availability of a machine of assembly language.

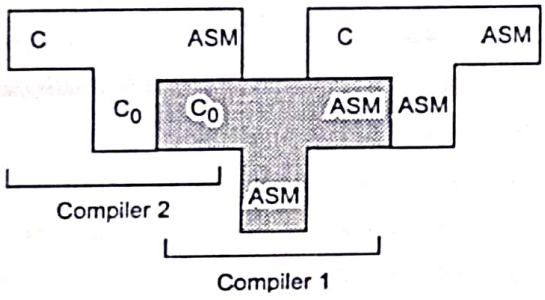
Step 1 : First we write a compiler for a small subset of C in assembly language.



Step 2 : Then using this small subset of C i.e. C_0, for the source language C the compiler is written.



Step 3 : Finally, we compile the second compiler. Using compiler 1 the compiler 2 is compiled.



Step 4 : Thus, we get a compiler written in ASM which compiles C and generates code in ASM.

PHASE OF COMPILER DESIGN

Q.11. Explain the analysis and synthesis model of compilation in brief. **CT : W-13(5M)**

OR Explain the advantages of analysis synthesis model of compilation. **CT : S-14(4M)**

Ans. The analysis synthesis model of compilation :

- There are two parts of compilation analysis and synthesis.
- The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The synthesis part constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialized techniques.
- During analysis, the operations implied by the source program are determined recorded in a hierarchical structure called a tree.
- Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation.

  For example, a syntax tree for an assignment statement is shown in following Fig.(a).

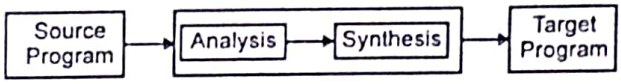


Fig.(a) Analysis and synthesis model

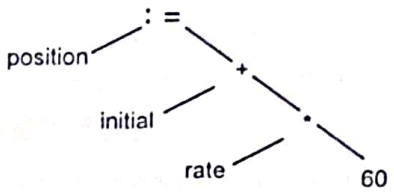


Fig.(b) Syntax tree for position : = initial + rate * 60.

- In compiling, analysis consist of three phases :

(1) Linear analysis : In this analysis, the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

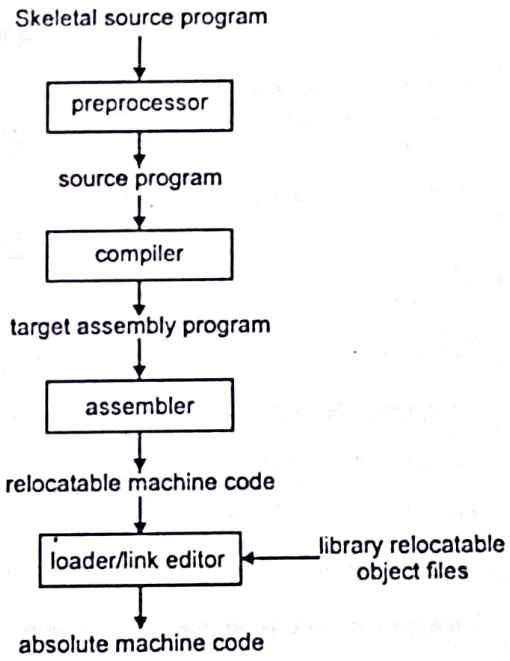


Fig.(c) A language processing system

(2) **Hierarchical analysis :** In this analysis, characters or tokens are grouped hierarchically into nested collections with collective meaning.

(3) **Semantic analysis :** In this analysis, certain checks are performed to ensure that the components of a program fit together meaningfully.

Advantages :

(i) Each phase has a well defined work.

(ii) Each phase handles a logical activity in the process of compilation.

(iii) Source and machine independent code optimization is possible.

(iv) Due to optimization phase front and back end phases can be deployed.

(v) Using this phases, compiler is retargetable i.e. compiler can be easily modified for another target machine or source language.

Q.12. Explain various phases of compiler in detail.

CS : S-11,12(8M), W-12(4M)

OR What are the different phases of compiler? Explain any one in detail.

CS : W-11(6M)

OR What are the various stages in the process of generation of target code from the source code by the compiler? How these stages are broadly classified? Explain each. **CT : S-14(10M)**

OR Explain each phase of compiler. Also discuss input required and output produced by each phase of compiler with suitable example.

CS : W-09(6M)

OR Explain complete compilation process for high level language program.

CT : W-10(4M)

OR Draw and explain general structure of compiler design.

CS : S-14(6M)

Ans. Phases of compiler :

- The process of compilation is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step.
- For this reason, it is customary to partition the compilation process into a series of sub-processes called phases as shown in Fig.(a).

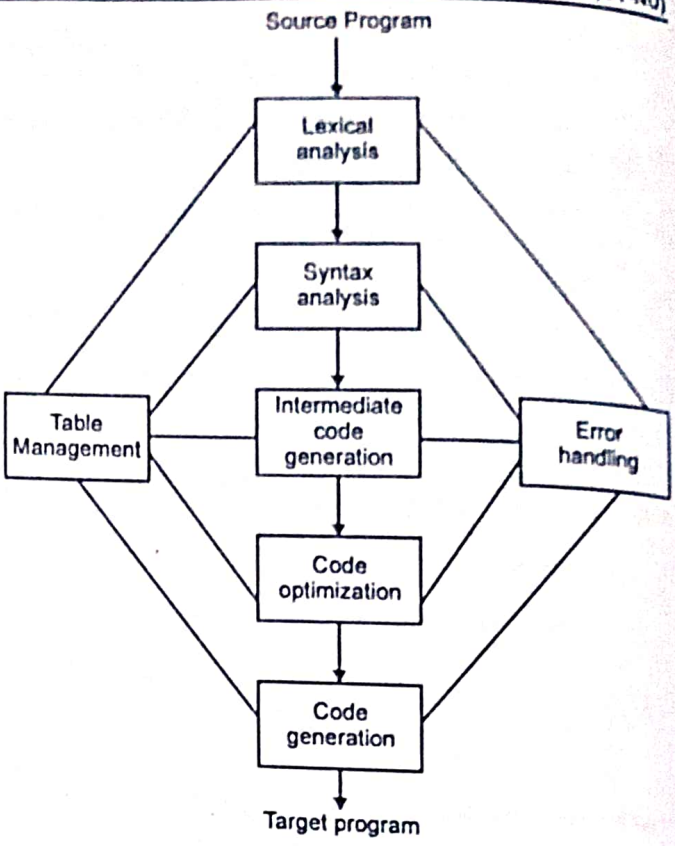


Fig.(a) Phases of compiler

(1) Lexical analysis :

- The lexical analysis is the interface between the source program and the compiler.
- The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of characters that can be treated as a single logical entity.
- Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. For example, in the FORTRAN statement.

  IF (5 . EQ. MAX); GOTO 100 .....('a')

  We find the following eight tokens :

  IF ; (; 5 ; EQ. ; MAX ;) GOTO ; 100
- What is called a token depends on the language at hand and to some extent on the discretion of the compiler designer, but in general each token is a substring of the source program that is to be treated as a single unit.
- There are two kinds of token; specific strings such as if or a semicolon and classes of strings such as identifiers, constants or labels.

- To handle both types of tokens, we shall treat a token as a pair consisting of two parts: a token type and a token value.
- A token such as the identifier MAX , above, has a type "identifier" and a value consisting of the string MAX.

Finding tokens :

- To find the next token, the lexical analyser examines successive characters in the source program, starting from the first character not yet grouped into a token.
- The lexical analyser may be required to search many characters beyond the next token in order to determine what the next token actually is.
- For example, Suppose the lexical analyser has last isolated the left parenthesis as a token in statement ('a'). We may represent the situation as follows:

IF (5 . EQ . MAX) GOTO 100

- Note that blanks have been removed, since they are ignored in FORTRAN.
- When the parser asks for the next token, the lexical analyzer reads all the characters between 5 and Q, inclusive, to determine the next token is just the constant 5.
- The reason it has to scan as far as it does is that until it sees the Q; it is not sure it has seen the complete constant ; it could be working on a floating-point constant such as 5-E-10.
- After determining that the next token is the constant 5, the lexical analyzer repositions its input pointer at the first dot, the character following the token.
- The lexical analyzer may return token type "constant" to the parser, and the value associated with this "constant" could be the numerical value 5 or a pointer to the string 5.
- When statement ('a') is completely processed by the lexical analyzer, the token stream might look like :

If ([const, 341] eq [id, 729] goto [label, 554])

- The relevant entries of the symbol table are suggested in Fig.

341	Constant, integer, value = 5
554	Label, value = 100
729	Variable, integer, value = MAX

Fig. (b) Symbol table

(2) Syntax analysis :

- The parser has two functions.
- It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.
- It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.
- For example, if a PL/I program contains the expression :

A + / B

Then 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 formation rules of a PL/I expression.
- The second aspect of 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 together.
- For example, the expression

A / B * C

has two possible interpretations :

(a) Divide A by B and then multiply by C ; or

(b) 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, a diagram which exhibits the syntactic structure of the expression.
- Parse tree that reflects orders (a) and (b) are shown in Fig. (c) and (d), respectively.

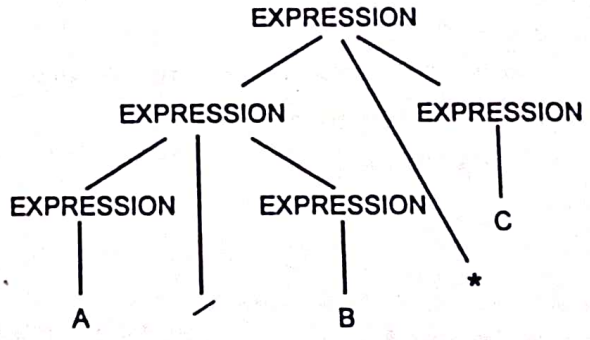


Fig.(c)

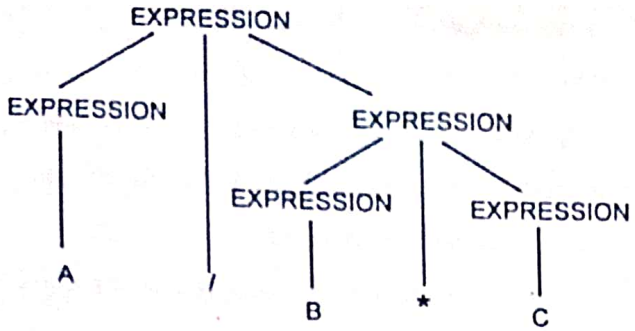


Fig.(d)

(3) Intermediate code generation :

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

Three-Address Code :

- A typical three- address code statement is A = B op C

  where A, B and C are operands and op is binary operator.
- The parse tree in Fig. (c) might be converted into the three address code sequence.

  $T_1 := A / B$

  $T_2 := T_1 * C$

  where, T_1 and T_2 are names of temporary variables.
- In addition to statements that use arithmetic operators, an intermediate-language needs unconditional and simple conditional branching statements, in which at most one relation is tested to determine, whether or not a branch is to be made.
- Higher-level flow of control statements such as while-do statements, or if then-else statements, are translated into these lower-level conditional three- address statements.
- For example, Consider the following while- statement

  While A > B & A <= 2 * B – 5 do

  A : = A + B

A straight forward algorithm for translation would produce intermediate code like that shown below :

L_1 : if A > B goto L_2

goto L_3

L_2 : $T_1 := 2 * B$

$T_2 := T_1 - 5 \rightarrow$

if $A \leq T_2$ goto L_4

goto L_3

L_4 : A := A + B

goto L_1

L_3 : Exit

(4) Optimization :

- Object programs that are frequently executed should be fast and small.
- Certain compilers have within them a optimization phase that tries to apply transformations to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object-language program can ultimately be produced.
- Optimizing compilers merely attempt to produce a better target program than would be produced with no "optimization".
- A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed, in comparison with a compiler that generates code carefully but without using the specialized techniques generally referred to as code optimization.
- There are two ways of optimization :

  (1) Local optimization

  (2) Loop optimization.

(5) Code Generation :

- The code generation phase converts the intermediate code into a sequence of machine instructions.
- A simple minded code generator might map the statement A := B + C into the machine code sequence.

  LOAD B

  ADD C

  STORE A
- However, such a straight-forward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores and that utilizes the resources of the target machine inefficiently.
- To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of the registers.
- Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary.

- Many computers have only a few high-speed registers in which computations can be performed particularly quickly.
- A good code generator would therefore attempt to utilize these registers as efficiently as possible. This aspect of code generation is called register allocation

(6) **Table management :**

- The table management, or book-keeping, portion of the compiler keep track of the names used by the program and records essential information about each, such as its type.
- The data structure used to record this information is called a symbol table.

(7) **Error handling :**

- One of the most important functions of compiler is the detection and reporting of errors in the source program.
- The error messages should allow the programmes to determine exactly whereby virtually all the phases of a compiler.
- A phase of the compiler discover an error; it must report the error to the error handler which issues an appropriate diagnostic message.

Q.13. What is front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into two distinct stages? **CT : W-11(6M),S-12(4M)**

OR Explain the structure of typical two pass compiler. Explain each component in detail. **CT : S-10(7M)**

Ans.

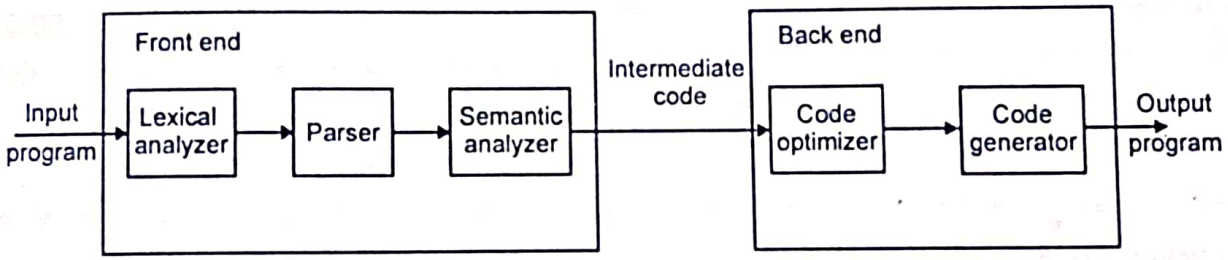


Fig. Front end back end model

- Different phases of compiler can be grouped together to form a front end and back end.
- The front end consists of those phase that primarily dependent on the source language and independent on the target language.
- The front end consists of analysis part. Typically it includes lexical analysis, syntax analysis, and semantic analysis. Some amount of code optimization can also be done at front end
- The back end consists of those phases that are totally dependent upon the target language and independent on the source language.
- In includes code generation and code optimization.
- The front end back end model of the compiler is very much advantageous because of following reasons :

(1) By keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.

(2) By keeping different front ends and same back end one can compile several different languages on the same machine.

Q.14. Discuss the factors which affect number of passes in compiler design. **CS : W-10(2M)**

Ans.

- Several phases of compiler are grouped into one pass. Thus pass of compiler is simply collection of various phases.
- For example, Lexical analysis, syntax analysis and intermediate code generation is grouped together and forms a single pass.
- Various factors affecting number of passes in compiler are :

  (i) Forward reference

  (ii) Storage limitation

  (iii) Optimization
- The compiler can require more than one passes to complete subsequent information.

Q.15. How can you differentiate between a phase and pass?

CS : W-10(3M)

OR What is difference between a phases and a pass of a compiler?

CT : S-12(4M)

Ans.

- In compilers, the process of compilation can be carried out with the help of various phases such as lexical analysis, syntax analysis, intermediate code generation, code generation and code optimization.
- Whereas in case of passes, various phases are combined together to form a single pass. An input program can be compiled using a single pass or using two passes.
- Different groups of phase form corresponding front end and back ends of the compilers. The advantage of front-end and back-end model of compiler is that : same source language can be compiled on different machines or several different languages can be compiled on the same machine.
- Compiling a source program into single pass is difficult because of forward reference that may occur in the program.
- Similarly if we group all the phases into a single pass then we may be forced to keep the entire program in the memory.
- And then memory requirement may be large. On the other hand it is desirable to have relatively few passes, because it becomes time consuming to read and write intermediate files.

Q.16. Discuss how the number of passes of compiler can be reduced.

Ans.

- It is desirable to have relatively few passes from the point of view of reducing the compilation time. And for reducing the number of passes, it is required to group several phases in one pass.
- For some of the passes, grouping into one pass is not a major problem.
- For example, The lexical analyzer and syntax analyzer, because the interphase between them is single token i.e. the processing required to be done with token is independent of other token.
- Therefore, these phases can be easily grouped together with lexical analyzer working as subroutine for syntax analyzer who is the in-charge of entire analysis activity.
- Whereas grouping of some of the phases in one pass is not easy e.g., grouping of intermediate code generation and target code generation is not easy because it is often very hard to perform code generation until the intermediate code has been generated because here the interface between the two is not in terms of only one intermediate instruction.
- This is because certain language permits use of variables before their declaration. Similarly, many languages permit forward jumps also.
- Therefore, it is not possible to generate target code for a construct until complete instruction is available to overcome this problem so as to enable the merging of intermediate code generation and target

Q.17. What are the various compiler writing tools? Explain each in brief. **CT : S-09(5M), CS : W-11(4M), S-14(7M)**

OR Explain in detail various compiler writing tools.

CT : S-10,13(6M)

OR Which tools are available to write a compiler? **CS : S-13(3M)**

OR Explain the useful compiler-construction tools.

Ans. Compiler-writing tools :

- The compiler writer, like any programmer, can profitably use software tools such as debuggers, version managers, profilers and so on.
- In addition to these software-development tools, other more specialized tools have been developed for implementation of various phases of a compiler.
- Some general tools have been created for the automatic design of specific compiler components. These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of a compiler.
- The following is a list of some useful compiler-construction tools

(1) Parser generators :

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

Q.15. How can you differentiate between a phase and pass?

CS : W-10(3M)

OR What is difference between a phases and a pass of a compiler?

CT : S-12(4M)

Ans.

- In compilers, the process of compilation can be carried out with the help of various phases such as lexical analysis, syntax analysis, intermediate code generation, code generation and code optimization.
- Whereas in case of passes, various phases are combined together to form a single pass. An input program can be compiled using a single pass or using two passes.
- Different groups of phase form corresponding front end and back ends of the compilers. The advantage of front-end and back-end model of compiler is that same source language can be compiled on different machines or several different languages can be compiled on the same machine.
- Compiling a source program into single pass is difficult because of forward reference that may occur in the program.
- Similarly if we group all the phases into a single pass then we may be forced to keep the entire program in the memory.
- And then memory requirement may be large. On the other hand it is desirable to have relatively few passes, because it becomes time consuming to read and write intermediate files.

Q.16. Discuss how the number of passes of compiler can be reduced.

Ans.

- It is desirable to have relatively few passes from the point of view of reducing the compilation time. And for reducing the number of passes, it is required to group several phases in one pass.
- For some of the passes, grouping into one pass is not a major problem.
- For example, The lexical analyzer and syntax analyzer, because the interphase between them is single token i.e. the processing required to be done with token is independent of other token.
- Therefore, these phases can be easily grouped together with lexical analyzer working as subroutine for syntax analyzer who is the in-charge of entire analysis activity.
- Whereas grouping of some of the phases in one pass is not easy e.g., grouping of intermediate code generation and target code generation is not easy because it is often very hard to perform code generation until the intermediate code has been generated because here the interface between the two is not in terms of only one intermediate instruction.
- This is because certain language permits use of variables before their declaration. Similarly, many languages permit forward jumps also.
- Therefore, it is not possible to generate target code for a construct until complete instruction is available to overcome this problem so as to enable the merging of intermediate code generation and target.

Q.17. What are the various compiler writing tools? Explain each in brief. **CT : S-09(5M), CS : W-11(4M), S-14(7M)**

OR Explain in detail various compiler writing tools.

CT : S-10,13(6M)

OR Which tools are available to write a compiler? **CS : S-13(3M)**

OR Explain the useful compiler-construction tools.

Ans. Compiler-writing tools :

- The compiler writer, like any programmer, can profitably use software tools such as debuggers, version managers, profilers and so on.
- In addition to these software-development tools, other more specialized tools have been developed for implementation of various phases of a compiler.
- Some general tools have been created for the automatic design of specific compiler components. These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of a compiler.
- The following is a list of some useful compiler-construction tools :

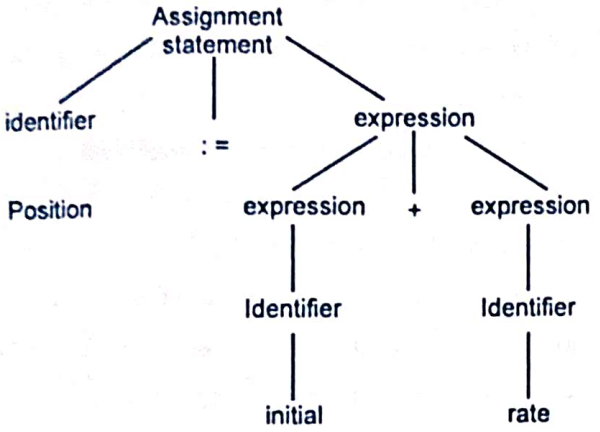
(1) Parser generators :

- These produce syntax analyzers, normally from input that is based on a context-free grammar.
- In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.

(2) **Scanner generators :** These automatically generate lexical analyzers, normally from a specification based on regular expression.

(3) **Syntax-directed translation engines :**

- These produce collection of routines that walk the parse tree, such as figure generating intermediate code.
- The basic idea is that one or more "translations" are associated with each node of the parse tree and each translation is defined in terms of translations at its neighbour nodes in the tree.



(4) **Automatic code generators :**

- Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
- The rules must include sufficient detail that we can handle the different possible access method for data : e.g., variables may be in registers, in a fixed (static) location in memory or may be allocated a position on a stack.
- The basic technique is "template matching". The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template.

(5) **Data-flow engines :**

- Much of the information needed to perform good code optimization involves "data-flow analysis", the gathering of information about how values are transmitted from one part of program to each other part.
- Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

Q.18. Why design of a compiler is complex than that of an assembler?

CS : S-09(3M), CT : S-09(2M)

OR Compiler is very complex to design than assembler. Justify..

CS : W-09(4M), CT : W-09(3M)

Ans.

- Compiler takes as its source code a C program i.e. a file of ASCII characters.
- It produces either an assembly language program, as an intermediate step or else machine code directly.
- The assembler takes as its source code an assembly language programme i.e. also final of ASCII character s, it used this to produce machine code.
- A compiler is definitely complex to write than assemblers. An assembler, in essence, is just a table that maps short strings (instruction mnemonics) to numbers (opcodes).
- The harder part of writing an assembler is just creating the table efficiently.
- A compiler on other hand, needs a much more complex passer (depending on the language) and needs to manage stack positions and register usage in arbitary circumstances.

Q.19. Explain the output of lexical analysis phase. Give the importance of this output in next phase of compiler. **CS : W-14(3M)**

Ans.

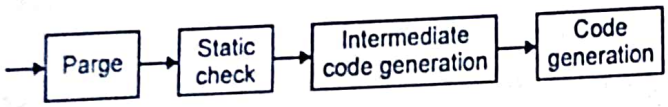
- The lexical analysis is the first phase which reads the character from the source program and group them into the stream of token.
- Token is logically cohesive sequence of characters such as identifiers, operators, keywords etc.
- A sequence forming a token is called lexeme.
- The output of lexical analyzer is a stream of token which is passed to the next phase i.e. syntax analysis.
- The syntax analyzer groups token together into syntactic structure.
- In next phase after lexical analysis, the compiler verifies, whether the tokens generated by lexical analyzer are grouped according the syntactic rules of the language or not.
- If they are grouped according to rules, then group of token generated by the lexical analyzer is a accepted or the valid construct of the language.

- The syntactic structure can be regarded as a tree whose leaves are the token.
- The interior nodes of the tree shows the string of token that logically belong together.

Q.20. What would happen if code generation phase is implemented after intermediate code generation?

Ans.

- Code generation phase generate the target code taking input as intermediate code generation phase. The output of intermediate code generation may be given directly to code generation or may pass through code optimization before generating code.
- Intermediate code is generated using the parse rule producing a language from the input language. Intermediate code has following property :
- Simple enough to be translated to assembly code.
- Complex enough to capture the complication of high level language.



- The input to code generation consist of intermediate representation of source program produced by front end, together with information in symbol table.
- It is used to determine run time address of data objects denoted by names in intermediate representation.
- So if code generation is implemented after intermediate code generation the intermediate code representation is not possible.
- So error removal process is not so efficient, so require more time for compilation.

Q.21. What is role of symbol table for each phase of compiler?

CT : IV-12(6M)

Ans.

- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
- These attributes may provide information about the storage allocated for an identifier, its type, its scope and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument and the type returned if any.
- A symbol table is a data structure containing the record of each identifier, with field for the attributes of identifier the data structure allows us to find the record for each identifier quickly and store or retrieve data from that record quickly.
- When ab in the source program is detected by the lexical analyzer the identifier is entered into the symbol table.
- However, the attributes of an identifier cannot normally be determined during lexical analysis.
- For example, in a pascal declaration like VAR position, initial, rate real ; the type real is not known when position, initial and rate are seen by the lexical analyzer.
- The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.
- For example, when doing semantic analysis and intermediate code generation, we need to know what the types of identifiers are, and so that we can check that the source program uses them in valid ways and so that we can generate the proper operations on them.
- The code generator typically enters and uses detained information about the storage assigned to identifiers.

Q.22. What are the errors that may encounter in each phase of a compiler?

CS : S-10.12(5M)

OR What kinds of errors may be generated by different phases of compiler?

Ans.

- The program contain errors at many different levels. For example error can be :

(1) Lexical, such as mis-spelling an identifier, keyword or operator.

(2) Syntactic, such as an arithmetic expression with unbalanced parentheses.

(3) Semantic, such as an operator applied to an incompatible operand.

(4) Logical, such as an infinitely recursive call.

- Often much of the error detection and recovery in a compiler is centered around the syntax analysis phase.
- One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analysis disobeys the grammatical rules defining the programming language.
- Another is the precision of modern parsing methods they can direct the presence of syntactic errors in programs very efficiently.

- Accurately detecting the presence of syntactic errors in program very efficiently. Accurately detecting the presence of semantic and logical errors at compile time is a much more difficult task.

DESIGN OF LEXICAL ANAYZER

Q.23. Explain the role of lexical analyzer in compiler designing.

OR Explain the block schematic of lexical analysis.

OR Discuss the issues in design of a lexical analyzer.

CS : W-10,13(4M)

Ans.

- Lexical analysis is a process of recognizing tokens from input source program.
- The lexical analyzer stores the input in a buffer. It builds the regular expressions for corresponding tokens. From these regular expressions, finite automata is built.
- When lexeme matches with the pattern generated by finite automata, the specific token gets recognized. The block schematic for this process is as shown below :

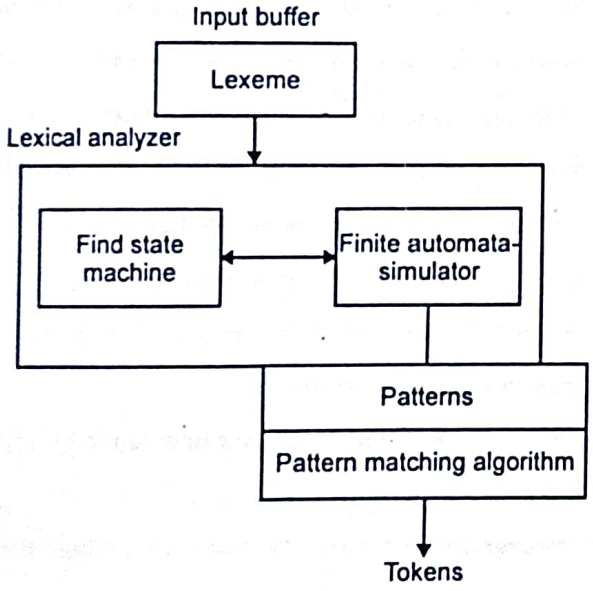


Fig.(a) Block schematic of lexical analyzer

- Lexical analyzer is the first phase of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens.
- Each token is a single logical cohesive unit such as identifier, keywords, operations and punctuation marks.
- Then the parser to determine the syntax of the source program can use these tokens.
- The role of lexical analyzer in the process of compilation is as shown below :

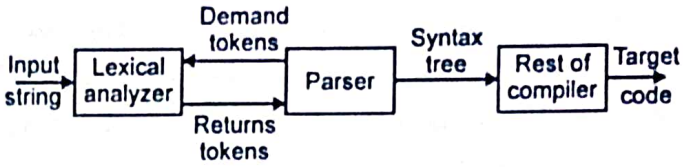


Fig.(b) Role of lexical analyzer

- As the lexical analyzer scans the source program to recognize the tokens it is also called as scanner. Apart from token identification lexical analyzer also performs following functions.

Functions of lexical analyzer :

(1) It produces stream of tokens.

(2) It eliminates blank and comments.

(3) It generates symbol table which stores the information about identifiers, constants encountered in the input.

(4) It keeps track of line numbers.

(5) It reports the error encountered while generating the tokens.

- The lexical analyzer works in two phases. In first phase it performs scan and in the second phase it does lexical analysis, means it generates the series of tokens.

Issues in lexical analysis :

There are several reason for separating the analysis phase of compiling into lexical analysis and parsing :

(i) Simpler design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one or other of these phases.

(ii) Compiler efficiency is improved.

(iii) Compiler portability is enhanced.

(iv) Identifying the tokens of the language for which lexical analyzer is to be built and to specify these tokens by using suitable notations.

Q.24. What is the role of input buffering scheme in lexical analyzer?

Ans.

- The lexical analyzer scans be characters of source program one by one to find the tokens.
- Moreover, it needs to look ahead several characters beyond the next token to determine the next token itself.

- So, an input buffer is needed by the lexical analyzer to read its input. In a case of large source program, significant amount of times is required to process the characters during the compilation.
- To reduce the amount of overhead needed to process a single character from input character stream, specialized buffering techniques have been developed.
- An important technique that uses two input buffers that are reloaded alternately is shown in Fig.

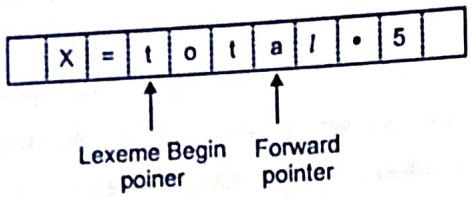


Fig. Input buffer

- Each buffer is of the same size N, where N is the size of a disk block, for example 1024 bytes. Thus, instead of one character, N characters can be read at a time.
- The pointers used in the input buffer for recognizing the lexeme are as follows :

(i) Pointer lexemeBegin points the beginning of the current lexeme being discovered.

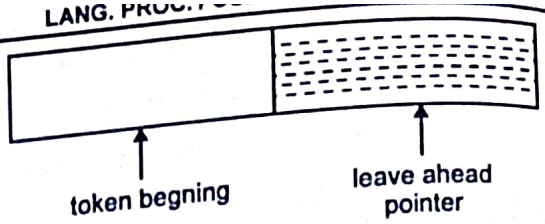
(ii) Pointer forward scans ahead until a pattern match is found for lexeme.

- Initially, both pointers point to the first character of the next lexeme to be found.
- The forward pointer is scanned ahead until a match for a pattern is found.
- After the lexeme is processed, both the pointers are set to the character following that processed lexeme.
- For example, in Fig. the lexemeBegin pointer is at character t and forward pointer is at character a. The forward pointer is scanned until the lexeme total is found. Once it is found, both these pointers point to*, which is the next lexeme to be discovered.

Q.25. Why lexical analyzer reads few characters beyond the token in advance before declaring validity of token? Explain with example. **CS : W-13(5M)**

Ans.

- The logical analyzer may be required to search many characters and the next token because sometimes logical analyzer is not sure that it has been complete token until it scans many characters beyond.



- If (5. EQ. MAX) GOTO 100.
- The string to left of arrow represents token by logical analyzer.
- When parser ask for next token logical analyzer read all character between 5 and a inclusive to determine that next token is just constant 5.

SCANNER GENERATOR : LEX, FLEX

Q.26. Write short note on scanner generator.

Ans. Scanner generator :

- These automatically generated lexical analyzer, normally form a specification based on regular expression.
- Several tools has been built for constructing lexical analyzer from special purpose notations based on regular expression. eg. lex tool.
- Lex compiler is input specification as the lex language.
- Lex like specification can be used even if a lex compiler is not available the specification can be manually transcribed into working program using the transition diagram techniques.
- A specification of a lexical analyzer is prepared by creating a program lex l in the lex language.
- Then lex. l is run through the lex compiler to produce a C program lex yy. C.
- The program lex.yy.C consist of a tabular representation of transition diagram constructed from regular expression of lex. l.

Q.27. Write a short note on LEX. **CS : W-14(2M)**

OR Explain Lex in details with the help of example.

OR Explain Lex.

OR What is LEX? **CS : S-10,11(1M), CT : W-09(2M)**

Ans. LEX :

- Lex is a program generator designed for lexical processing of character input streams.

- It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.
- Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages".
- Lex can write code in different host languages. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users.
- Lex turns the user's expressions and actions into the host general-purpose language; the generated program is named yylex.
- The yylex program will recognize expressions in a stream and perform the specified actions for each expression as it is detected.

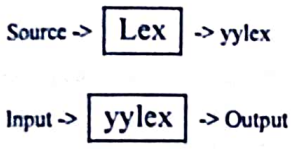


Fig. An overview of Lex

- Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex.
- Lex generates a deterministic finite automata from the regular expressions in the source.
- Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.
- Lex source is a table of regular expressions and corresponding program fragments.
- The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions.
- As each such string is recognized the corresponding program fragment is executed.
- The recognition of the expressions is performed by a deterministic finite automata generated by Lex.
- The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

Structure of a Lex :

The structure of a Lex file is intentionally similar to that of a YACC file; files are divided into three sections, separated by lines, as follows :

(1) **Definition Section :** The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

(2) **Rules Section :** The rules section associates regular expression patterns with C statements. When the lexer identifies text in the input matching a given pattern, it will execute the associated C code.

(3) **C code section :** The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Q.28. Describe a tool for study of Lex.

OR Explain any one tool for writing lexical analyzer. *CS : S-13(3M)*

Ans.

- Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
- A tool, called Lex that has been widely used to specify lexical analyzers for a variety of languages, refer to the tool as Lex compiler, and its input specification as the Lex language.
- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l, together with a standard routine that uses the table to recognize lexemes.
- The actions associated with regular expression in lex.l are pieces of C code and are carried over directly to lex.yy.c. Finally lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

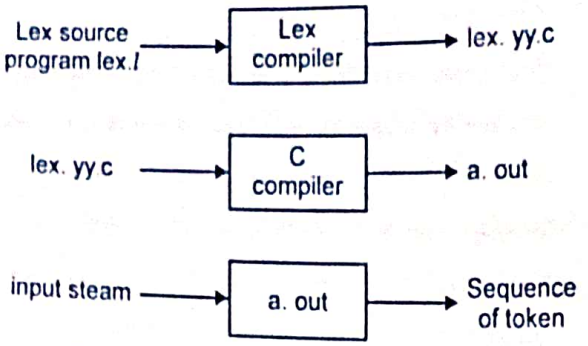


Fig. Creating a lexical analyzer with lex

Q.29. Give general specification of LEX program of its compiling process. **CT : W-09(2M)**

OR Give specification of LEX program. **CS : S-10(2M)**

OR Explain in brief, the format of LEX input file. **CT : S-10(3M)**

OR Explain structure of LEX. **CS : S-11(2M)**

OR Write a LEX program to produce a lexical analyzer that recognizes identifiers. **CT : S-11(6M)**

Ans.

- LPDT (Language Processor Development Tools) generates the analysis phase of a language processor whose source language is L.
- The LPDT requires the following two inputs :

(1) Specification of a grammar of language L.

(2) Specification of semantic actions to be performed in the analysis phase.

- It generates programs that perform lexical, syntax and semantic analysis of the source program and construct the IR.
- These programs collectively form the analysis phase of the language processor.
- These are the lexical analyzer generator LEX and the parse generator YACC.
- The input to these tools is a specification of the lexical and syntactic constructs of L, and the semantic actions to be performed on recognizing the constructs.
- The specification consists of a set of translation rules of the form

<string specification> {<semantic action>}

where <semantic action> consists of C code.

- This code is executed when a string matching <string specification> is encountered in the input.
- LEX and YACC generate C programs which contain the code for scanning and parsing, respectively, and the semantic actions contained in the specification.

- Figure shows a schematic for developing the analysis phase of a compiler or language L using LEX and YACC.

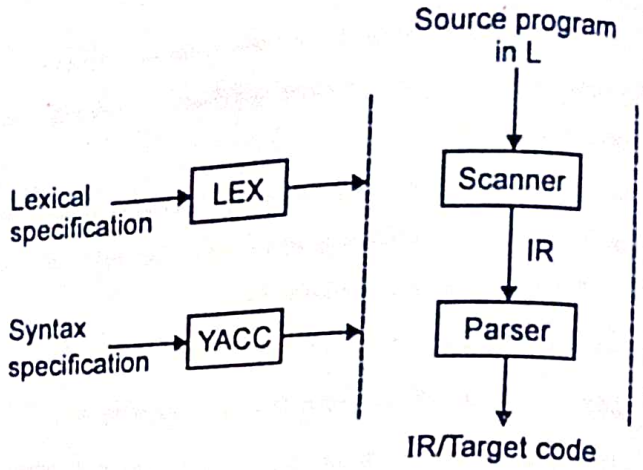


Fig.(a) Using LEX and YACC

LEX :

- LEX accepts an input specification which consists of two components.
- The first component is a specification of strings representing the lexical units in L, e.g. id's and constants.
- This specification is in the form of regular expression.
- The second component is a specification of semantic actions aimed at building an IR.

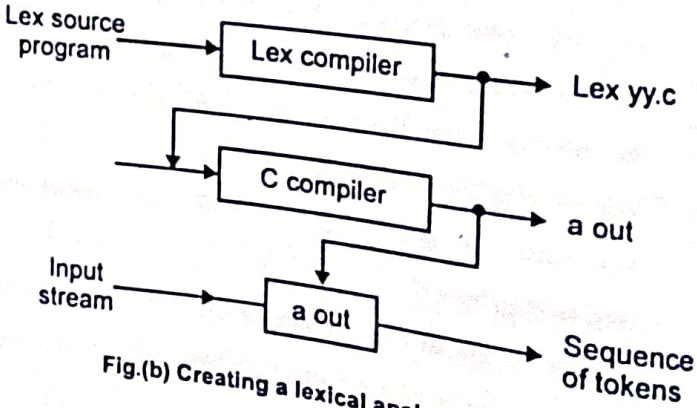


Fig.(b) Creating a lexical analyser with lex

- LEX has been widely used to specify lexical analyzer for variety of languages. Tool has Lex compiler so its input specification has Lex language.
- Lex is a software that takes as input specification of a set of regular expression together with actions to be taken.
- The output of Lex is a program that recognizes the regular expression and acts appropriately on each.

Structure of Lex Programs :

The general form of a lex program like lex.1 is

declarations

%%

translation rules

%%

auxiliary functions

lx program to recognize identifier name, integer constant, floating point constant and arithmetic operator is as follows :

```
%{
/*Literal block*/
#include "scan.h" /*Scanner header file*/
int line = 1; /*Current line number*/
int pos = 0, /*Start position of token*/
int epos = 1; /*End position of token*/
OpType op; /*Last operator scanned*/
void lex_err(char *, char *); /*Reports lexical errors*/
%}
/*Regular definitions*/
letter          [A-Za-z]/*letter can be underscore*/
digit           [0-9]
blank_str       [ \t]+
identifier      ({letter} ({letter} | {digit}))*
int_literal     {digit}+str_literal \"([^\"\n])*\"
/*Regular definitions to match invalid tokens*/
open_string     \"({^\"\n])*$
%% /*Second section*/
\n {line++; epos = 1;}
{blank_str} {epos +=yyleng;}
[Ii][Ff] {pos = epos; epos += 2; return IF_TK;}
[Tt][Hh][Ee][Nn] {pos = epos; epos +=4; return THEN_TK;}
[Ee][Ll][Ss][Ee] {pos = epos; epos +=4; return ELSE_TK;}
[Ee][Nn][Dd] {pos = epos; epos +=3; return
END_TK;}
[Ww][Hh][Ii][Ll][Ee] {pos = epos; epos +=5;
return WHILE_TK;}
[Dd][Oo]        {pos = epos; epos +=2; return
DO_TK;}
[Rr][Ee][Aa][Dd] {pos = epos; epos +=4; return READ_TK;}
[Ww][Rr][Ii][Tt][Ee] {pos = epos; epos +=5; return WRITE_TK;}
{identifier} {pos = epos; epos += yyleng; return ID;}
{str_literal} {pos = epos; epos += yylen; return STRLIT;}
{int_literal} {pos = epos; epos += yyleng; return INTLIT;}
/*Regular definition to identify arithmetic operator*/
"+" {op = ADD, pos = epos; epos += 1; return ADDOP;}
"-" {op = SUB; pos = epos; epos += 1; return ADDOP;}
"*" {op = MUL; pos = epos; epos += 1; return MULOP;}
"/" {op = DIV; pos = epos; epos += 1; return MULOP;}
"=" {op = EQ; pos = epos; epos += 1; return RELOP;}
"<>" {op = NE; pos = epos; epos += 2; return RELOP;}
"<" {op = LT; pos = epos; epos += 1; return RELOP;}
"<=" {op = LE; pos = epos; epos += 2; return RELOP;}
">" {op = GT; pos = epos; epos += 1; return RELOP;}
">=" {op = GE; pos = epos; epos += 2; return RELOP;}
":=" {op = ASGN; pos = epos; epos += 2; return ASGNOP;}
"," {pos = epos; epos += 1; return ' , ';}
";" {pos = epos; epos += 1; return ' ; ';}
"(" {pos = epos; epos += 1; return ' ( ';}
")" {pos = epos; epos += 1; return ' ) ';}
/*Regular definition to indentify Unsigned integer or floating point
numbers*/
        digit → [0-9]
        digits → digit
        number → digits (. digits)?(E+-]?digits)?
}
```

Q.30. Explain the regular expressions in LEX.

Ans. Regular expressions in LEX :

- In Lex, the regular expression is a pattern description using a meta language, a language that we use to describe particular patterns of interest.
- The characters used in this meta language are part of the standard ASCII character list.
- The characters that form regular expression are :

cter	Description
.	matches any single ASCII character except the new line character ("\n").
*	matches zero or more copies of the preceding expression or character.
[ ]	This is to represent the character class that matches any character within the brackets.
^	If any character set or expression comes just after the operator, it will accept all the characters except the one within the character class, that is, this character can be used to negate any character class within the square brackets .
–	A dash inside the square brackets indicates a character range, for example [0 – 9] means the same thing as [0123456789] and [A – Z] means all the English alphabet letters A to Z.
{ }	It indicates how many times the previous pattern is allowed to match which contains one or two number. For example, A {1,3} matches one to three occurrences of the letter A.
\	It is used to escape meta-characters and as part of the usual C escape sequence, for example, "\n" is a new line character, while "*" is literal asterisk.
+	Matches one or more characters of the preceding regular expressions or a character set. For example, [0 – 9] + matches one or more combinations of the character 0 to 9. This is, this regular expression accepts the strings 0112, 1231456, or 9012, but not an empty string.
?	It matches zero or one occurrence of the preceding regular expression, for example, – [0 – 9] + matches a signed number including an optional leading minus.
(10)	|	Matches any one of the preceding or following regular expressions, for example, pen/pencil/eraser. This expression will match any one of the three words.
(11)	"..."	It interprets everything within the equation marks as meta-characters other than the C escape sequences.
(12)	()	Groups a series of regular expressions together into a new expression. For example, (10101) represent a character sequence 10101.
(13)	<>	A name or a list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.
(14)	<<EOF>>	In Flex, the special pattern matches the end of the file.

Q.31. Write regular expression for signed integer constant.

CS : S-11(2M)

Ans. Regular expression for signed integer constant :

The exponential form of real number is a decimal or signed integer followed by E and signed integer i.e.,

Exponential → (decimal / signed integer) E

signed integer

Sign → + / – / E

Digit → 0 / 1 / ..... / 8 / 9

Integer → digit (digit) *

Floating → integer / E

signed - integer → Sign integer

Decimal → Sign - integer floating / signal integer.

Q.32. What is FLEX?

OR Explain FLEX in details.

Ans. FLEX :

- Flex is a fast lexical analyzer generator tool for programming that recognizes lexical patterns in the input with the help of Flex specifications.
- Flex specification contains two parts :

  (i) patterns and (ii) corresponding action.

- When we write a flex specification, we create a set of patterns which the lexer matches against the input.
- Each time one of the patterns matches, the corresponding action part is invoked (which is a C code). In this way, a Lex program divides the input into tokens.
- Flex itself does not produce an executable program, instead, it translates the Lex specifications into a file containing a C subroutine called yylex().
- More precisely, all the rules in the rules section will automatically be converted into the C statements by the Flex tool and will be put under the function name of yylex(). That is, whenever we call the function yylex, C statements corresponding to the rules will be executed.
- That is, we call yylex() to run the lexer. The generated lexical analyzer, by default it is lex.yy.c, can be compiled using regular C compiler along with any other files and Flex libraries.
- The program that we write in a Lex program contains the Lex specification and other C statements and subroutines. This file is named with an extension <filename>. *l* (for example, file.*l*).
- When this Lex program is passed to the Flex, it translates the <filename>. *l* into a file named lex.yy.c, which is a C program. Fig. shows the phases of a Lexical analyzer.

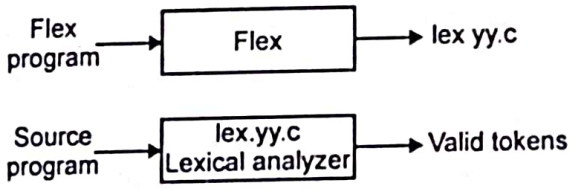


Fig. Phases of a lexical analyzer.

- The file lex.yy.c is also called the lexer or lexical analyzer.
- This C program is compiled with a normal C compiler and it produces an executable program. This can be executed like a normal C executable file.

Q.33. Give the structure of FLEX program.

Ans. Structure of FLEX program :

- Any Flex program consists of three sections separated by a line with just %% in it.

Definition section

%%

Rules section

%%

User code (Auxiliary) section

The basic structure of a Lex program. It consists of three sections

(i) A definition section

(ii) A rules section

(iii) A user defined section

- The first section, i.e. the definition section contains different user defined Lex options used by the lexer. It also creates an environment for the execution of the Lex program.
- The definition section helps to create an atmosphere in two areas. First, it creates an environment for the lexer, which is a code.
- This area of the Lex specification is separated by " % {" and " %}", and it contains C statements, such as global declarations, commands, including library files and other declarations, which will be copied to the lexical analyzer (i.e. lex.yy.c) when it is passed through the Flex tool.
- In other words, Flex copies all the statements in this C declaration section bracketed by "% {" and " %}" to the lexical analyzer file, which is lex.yy.c.
- Secondly, the definition section provides an environment for the Flex tool to convert the Lex specifications correctly and efficiently to a lexical analyzer.
- This section mainly contains declarations of simple name definitions to simplify the scanner specifications and declarations of start condition. The statements in this section will help the Lex rules to run efficiently.
- The second section of any Lex program is the rules section that contains the patterns and actions that specify the Lex specifications.
- A pattern is in the form of a regular expression to match the largest possible string. Once the pattern is matched, the corresponding action part is invoked.
- The action part contains normal C language statements(s). They are enclosed within braces (i.e. "{" and "}"), if there is more than one statement, to make these component statements into a single block of statement.
- The Lex tools take everything after the pattern to be the action, while others only read the first statement on the line and silently ignore other statements. Always use braces to make the code clear, if the action has more than one statement or more than one line large.

- The lexer always tries to match the largest possible string, but when there are two possible rules that match the same length, the lexer uses the first rule in the Lex specification to invoke its corresponding action.
- The third and the final section of the Lex program is the user defined or user subroutine section. It is also known as auxiliary section.
- This section contains any valid C code. Flex copies the contents of this section into the generated lexical analyzer as it is. The simplest Flex program is :

  %%

  which generates a scanner that simply copies its input (one character at a time) to its output.

PROGRAMMING LANGUAGE GRAMMAR

Q.34. What are the elements to be considered while designing a new language processor?

OR Discuss salient features of a good programming language.

OR Write down reasons to explain why programmers prefer one language over another.

OR Give the features of High-level programming language.

OR What are the features of programming language?

Ans.

- A programming language is a notation with which people can communicate algorithms to computers and to one another. Hundreds of programming languages exist. They differ in their degree of closeness to natural or mathematical language one hand and to machine language on the other. They differ also in the type of problem for which they are best suited.
- Some of the aspects of high-level languages which make them preferable to machine or assembly languages are the following :

(1) Ease of understanding :

- A high-level language program is generally easier to read, write and prove correct than is an assembly- language program because a high-level language usually provides a more natural notation for describing algorithms than does assembly language.
- Even high level languages, some are easy to use than others.
- A good programming language should provide features for modular design of easy to understand programs. A good language should also enable control flow to be specified in a clean understandable manner.

(2) Naturalness : Much of the understandability of high ... programming language comes from the ease with which one ... express an algorithm in that language.

(3) Portability : Users must often be able to run their programs on ... variety of machines. Languages such as FORTRAN or COBOL ... have relatively well-defined "standard version", and programs ... conforming to the standard should run on any machine.

(4) Efficiency of use :

- This area cover a number of aspects of both program and language ... design. One would like to be able to translate source program into ... efficient object code. One would also like the compilation process ... itself to be efficient.
- It is often more important that the programmer be able to ... implemented programs in a way that makes efficient use of ... time.
- A high level programming language should have facilities for ... defining data structures, macros, subroutines, etc.
- Particular high-level language features facilitate reliable ... programming. Some such features are :

(a) Data structures.

(b) Scope rules that allow modifications to be made to a piece of a program without unwittingly affecting other portions of the same program.

(c) Flow-of-control constructs that clearly identify the looping structures in a program.

(d) Subroutines, allowing a program to be designed in small pieces.

Q.35. Explain the following terms :

(i) Syntax and semantics.

(ii) Source code and object code. CS : W-14(10)

OR Compare and contrast :

(i) Syntax and Semantics

(ii) Source code and Object code.

Ans.

(i) Syntax and Semantics :

- A program in any programming language can be viewed as string of symbols or characters choosen from some alphabet.
- But any arbitrary string of characters from the choosen alphabet does not represent a valid program.

- The rules telling whether a string is a valid program or not are called the syntax of the programming language.
- Certain notations, namely regular expressions and context-free grammars, are useful only for specifying much of the syntax of programming language, but also for helping in the construction of their compilers.
- If a program is valid program, then it is essential to know what a program means for it's faithful compilation into machine language program that will do what the programmer expects.
- The rules that give meaning to program are called the Semantics of a programming language is much harder to specify than it's syntax.

(ii) Source code and Object code :

- A compiler is a program which performs a job of translating a program written in high-level language into an equivalent program in a language close to machine.
- The program form written by the programmer using high-level language which is input to the compiler is called as Source code.
- The output program constructed by the compiler is as object code.
- For example, A program written in Pascal is source code and produced by Pascal compiler when this program is given as input is Object code.

Q.36. For a string S, define the following terms with an example :

(1) Prefix (2) Suffix

(3) Substring (4) Proper suffix

(5) Proper prefix (6) Length of string

(7) Language.

Ans.

(1) Prefix :

- If S is some string, then any string formed by discarding zero or more trailing symbols of s is called prefix of S.
- For example, abc is a prefix of abcde

(2) Suffix :

- If S is some string, then any string formed by deleting zero or more of the leading symbols of S is called suffix of S.
- For example; cde is suffix of abcde.

(3) Substring :

- A substring of S is any string obtained by deleting a prefix and a suffix from S. Moreover, any prefix or suffix of S is a substring of S, but a substring need not be a prefix or suffix.
- For example, cd is substring of abcde but not a prefix or suffix.

(4) Proper suffix : Let W = xy be a string. Then 'y' is a proper tail or suffix, if x is non empty.

(5) Proper prefix : Let W= xy be a string. Then 'x' is a proper prefix, if y is non empty.

(6) Length of string :

- If W is the string, then length of string is denoted as | W| and it is count of no. of symbols of which W is made up of.
- For example, If W = xyz, then | W| = 3. If | W | = 0, then the string is called as empty string and we use $\in$ to denote empty string.

(7) Language :

- It is a set of string formed from some specific alphabet.
- Simple sets such as the empty set, having no member or $\{\in\}$, the set containing only the empty string, are languages under this definition.
- If L and M are language then L.M or LM is the languages consisting of all strings xy which can be found by selecting a string x from L, a string y from M, and concatenating them in that order.

  LM = { xy / x is in L and y is in M}
- For example, L = {a, b} M = { c, d}

  then Reduce LM = { ac, ad, bc, bd }.

(8) Alphabet : It is finite set of symbols denoted by letter $\sum$.

(9) Production :

- A production of a grammar is a rule relating the variable.
- A typical production states that the language associated with a given variables contains strings from the language of certain other variables possibly along with some terminals.

Q.37. Discuss various parameter passing techniques with suitable example.

OR Explain with example parameter passing techniques used in programming languages. **CT : S-12(6M)**

OR Write short note on parameter transmission. **CT : W-12(4M)**

OR Write short note on parameter passing.

Ans. Parameter passing :

- When one procedure calls another, the usual method of communication between them is through nonlocal names and through parameters of the called procedure.

- Both nonlocals and parameters used by the procedure is to exchange the values of a[i] and a[j]. Here, array a is nonlocal to the procedure exchange and i and j are parameters.

  For example, consider a pascal procedure swap with nonlocal and parameters.

  (1) procedure exchange (i, j : integer) ;

  (2) var x : integer

  (3) begin

  (4) x := a[i] ; a[i] : = a [j] ; a[j] : = x;

  (5) end

- There are several common methods for associating actual and formal parameters. They are : call by value, call by reference, copy restore, call- by name and macro expansion.

(I) Call-by-value :

- This is the simplest method of passing parameters.
- The actual parameters are evaluated and their r-values are passed to the called procedure.
- Call by value is used in C and pascal parameters are usually passed this way.

Example :

Consider a pascal program with procedure swap.

(1) program reference (input, output);

(2) var a, b : integer;

(3) procedure swap (Var x, y : integer);

(4) begin

(5) temp := x;

(6) x = y ;

(7) y := temp;

(8) end;

(9) begin

(10) a : = 1; b : = 2;

(11) swap (a, b);

(12) writeln (a'=',a) ; writeln (b'=', b);

(13) end.

(II) Call-by reference :

- When parameters are passed by reference, the caller passes to the called procedure a pointer to the storage address of each actual parameter.

(1) If an actual parameter is a name or an expression having an l-value, then that l-value itself is passed.

(2) However, the actual parameter is an expression, like a + b or 2, that has no.l value then the expression is evaluated in a new location and the address of that location is passed.

- A reference to a formal parameter in the called procedure becomes, in the target code, an indirect reference through the pointer passed to the called procedure.

Example :

Consider a program with swap function using call by reference :

(1) swap (x, y)

(2) int *x, *y:

(3) { int temp;

(4) temp=*x ; *x =*y ; *y = temp;

(5) }

(6) main ( )

(7) { int a = 1, b = 2;

(8) swap (&a, &b);

(9) printf ("a is now %d, b is now %a\n", a, b);

(10) }

(III) Copy restore :

A hybrid between call-by-value and call-by-reference is copy-restore linkage.

Example :

Consider a simple program denostrate the use of copy restore.

(1) program copyout (input, output) ;

(2) var a : integer;

(3) procedure unsafe (vax x : integer);

(4) begin x : = 2; a : = 0 ;

(5) begin

(6) a :=1, unsafe (a); writeln (a);

(7) end.

(IV) Call-by-name :

Call-by-name is traditionally defined by the copy-rule of Algol which is :

(1) The procedure is treated as if it were a macro; that is, its body is substituted for the call in the caller, with actual parameters literally substituted for the formals. Such a literal substitution is called macro expansion or in-line expansion.

(2) The local names of the called procedure are kept distinct from the names of the calling procedure.

(3) The actual parameters are surrounded by parentheses if necessary to preserve their integrity.

Q.38. How parameter passing affects working of subroutine?

Ans.

- In subroutine (subprogram) formal parameter i.e. local data with subprogram is called actual parameter may be called in subprogram i.e. data object which is shared with the other subprogram.
- When a subprogram transfers control to another subprogram, there must be an association of the actual parameter of the calling subprogram with the formal parameter of the called program.
- Two approaches are often used; The actual parameter may be evaluated and that value passed to the formal parameter, or the actual object may be passed to the formal parameter.
- Whenever a formal parameter corresponding to a by name actual parameter is referenced in a subroutine, the thunk compiled for that parameter is executed.
- During call by reference, in execution of subprogram references to formal parameter names are treated as ordinary local variable references.
- During call by value, the alias to the actual parameter is created and that to be passed to subprogram execution program.

Q.39. Explain with suitable examples, the concept of *l*-value and r-value of an expression. **CT : IV-13(4M)**

Ans. L-value and r-value :

- There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments.

  i : = S ;

  i : = i + 1;

- The right side specifies an integer value, while the left side specifies where the value is to be stored. Similarly, if p and q are pointers to characters.

  $p^{\wedge}$: = $q^{\wedge}$;

- The right side $q^{\wedge}$ specifies a character, while $p^{\wedge}$ specifies where the character is to be stored. The term *l*-value and r-value after values that are appropriate on the left and right sides of an assignment, respectively. That is r-values are what we usually think of as "values," while *l*-values are locations.

Q.40. Explain significance of L and R value in writing assignment statements for a particular progra-mming languages. **CT : S-10(4M)**

Ans. Significance of *l*-and r-values:

- Here *l*-value means left value and r-value means right value. The expression has the following significance of *l*-and r-values :

(1) Expression with single name has same *l*- and r-value as that of name.

(2) Expression has a *l*-value if it denotes location.

Examples :

(a) Every name has an *l*-value, n-namely the location or locations reserved for its value.

(b) If A is an array name, the *l*-value of A [I] is the location/locations reserved for the I^{th} element of the array. The r-value of A [I] is the value stored there.

(c) Constant has an r-value but no *l*-value.

(d) If P is a points, its r-value is the location to which P points and its *l*-value is the location in which the value of P itself is stored.

REGULAR LANGUAGES

Q.41. Give formal definition of Regular language.

Ans. Regular language :

- The collection of regular languages over an alphabet Σ is defined recursively as follows :

(i) The empty language ϕ is a regular language.

(ii) For each $a \in \Sigma$, the singleton language {a} is a regular language.

(iii) If A and B are regular languages, then $A \cup B$ (union), $A \cdot B$ (concatenation) and A^* are regular languages.

(iv) No other languages over Σ are regular.

Example : All finite languages are regular, in particular the empty string language $\{\epsilon\} = \phi^*$ is regular.

- A simple example of a language that is not regular is the set of strings $\{a^n b^n \mid n \geq 0\}$.
- It cannot be recognized with a finite automata, since a finite automata has finite memory and it cannot remember the exact number of a's.

Q.42. Mention the properties of regular language.

Ans. In theoretical computer science, a regular language is a formal language that satisfies the following equivalent properties :

(i) It can be accepted by a deterministic finite state machine.

(ii) It can be accepted by a non-deterministic finite state machine.

(iii) It can be accepted by an alternating finite automation.

(iv) It can be described by a formal regular expression.

(v) It can be generated by regular grammar and prefix grammar.

(vi) It can be accepted by a read-only Turing machine.

FINITE AUTOMATA

Q.43. Define and explain finite Automata.

Ans. **Finite Automata :**

- Finite Automata is a basic model of computation, which will translate the I/P into O/P without any human interaction.
- It is a mathematical model of finite state machine. It consists of finite sets of states, I/P symbols and set of transitions.
- In finite state machine the internal state of machine alters when the required machine receives an I/P and generates the O/P.
- It depends upon present state and I/P and generates the O/P.
- It depends upon present state and I/P applied, the O/P generated is the result generated after the process satisfying certain characteristics.

Formal Definition :

Finite Automata is denoted by five tuple :

$M = (Q, \Sigma, \delta, q_0, F)$

where,

$Q \rightarrow$ Set of states, $(q_0, \ldots\ldots q_n)$.

$\Sigma \rightarrow$ Set of I/P symbols or alphabet.

$\delta \rightarrow$ Set of transitions, $Q \times \Sigma \rightarrow Q$.

$q_0 \rightarrow$ Initial state, $q_0 \subseteq Q$.

$F \rightarrow$ Set of final state, $F \subseteq Q$.

Capability of Finite Automata :

- It is used to decide whether the input is accepted or not accepted.
- If the machine terminates with final states the I/P is accepted.
- There are two types of Finite Automata (FA) :

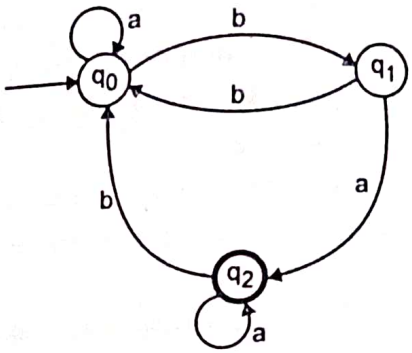
(1) Deterministic Finite Automata (DFA).

(2) Non Deterministic Finite Automata (NFA).

(1) Deterministic Finite Automata (DFA) :

If there is a single (unique) transition from one state to other with a single I/P symbol, then that automata is called as Deterministic Finite Automata.

Example :



A deterministic finite automata is a unituple.

$M = \{Q, \Sigma, \delta, q_0, F\}$

$M = \{(q_0, q_1, q_2), (a, b), \delta, (q_0), (q_2)\}$

Here, every state (q_0, q_1, q_2) for Input symbol (a,b), performs unique transition, like transitions are:

(1) $\delta(q_0, a) \rightarrow q_0$

(2) $\delta(q_0, b) \rightarrow q_1$

(3) $\delta(q_1, a) \rightarrow q_2$

(4) $\delta(q_1, b) \rightarrow q_0$

(5) $\delta(q_2, a) \rightarrow q_2$

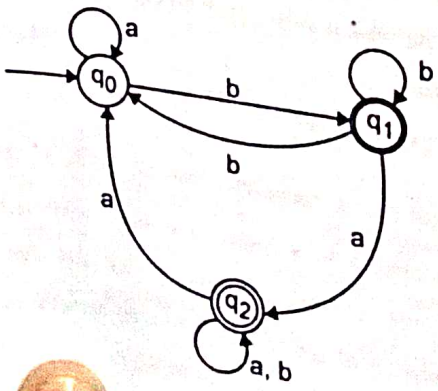
(6) $\delta(q_2, b) \rightarrow q_0$

} Unique transitions

(2) Non Deterministic Finite Automata (NFA) :

If there are more than one transitions from one state to another state with a single I/P symbol, then that automata is called as Non-Deterministic Finite Automata.

Example :



$M = \{(q_0, q_1, q_2), (a, b), \delta, (q_0), (q_1)\}$

Here, every state (q_0, q_1, q_2) for Input symbol (a, b), performs more than one transition shown as below :

(i) $\delta(q_0, a) \rightarrow \{q_0, q_2\}$

$\delta(q_0, b) \rightarrow \{q_1\}$

(ii) $\delta(q_1, a) \rightarrow \{q_2\}$

$\delta(q_1, b) \rightarrow \{q_0, q_1\}$

(iii) $\delta(q_2, a) \rightarrow \{q_2\}$

$\delta(q_2, b) \rightarrow \{q_2\}$.

Q.44. Give various application of finite automata.

OR Explain briefly application of finite automata.

OR Explain with suitable example the application of finite automata.

Ans. Application of Finite Automata are as follows :

(1) Lexical Analyzers.

(2) Text Editors.

(3) Switching circuit design.

(4) Text processing program.

(1) Lexical Analyzers :

- The tokens of programming languages can be expressed as regular sets.

  Example : ALGOL identifiers which are upper or lower - case letters followed by any sequence of letters and digits with no limit on length, may be expressed as :

  (letter) (letter/digit)*

- A number of lexical-analyzer generators take as input, a sequence of regular expressions describing the tokens and produce a single finite automaton recognizing any token.
- Usually, they convert the regular expression to an NFA with transitions and then construct subsets of states to produce a DFA directly, rather than first eliminating $\in$-transitions.
- Each final state indicates the particular token found, so the automaton is really a Moore machine.

(2) Text Editors :

Certain text editors and similar programs permit the substitution of a string for any string matching a given regular expression.

Example :

- The UNIX text editor allows a command such as : s/ bbb */b/ that substitutes a single blank for the first string of two or more blanks found in a given line.
- Let "any" denote the expression $a_1 + a_2 + ... + a_n$ where a_i's are all of a computers characters except the "new line" character.
- We could convert a regular expression r to a DFA that accepts any*r.
- The UNIX text editor converts the regular expression any* r is converted to an NFA with transitions and the NFA is then simulated directly.

Q.45. Discuss the role of finite automata in compiler.

Ans. Role of finite automata :

- The finite automata are used as the mathematical model that can be used to recognize the regular expressions.
- The regular expressions are built to match the pattern of the lexeme to identify the tokens. In order to recognize tokens the regular expressions can be created.
- These regular expressions then can be converted into the equivalent NFA. The NFA then can be converted to DFA.
- Input string is read character by character and lexical analysis is done using the finite automata so that valid tokens can be generated.

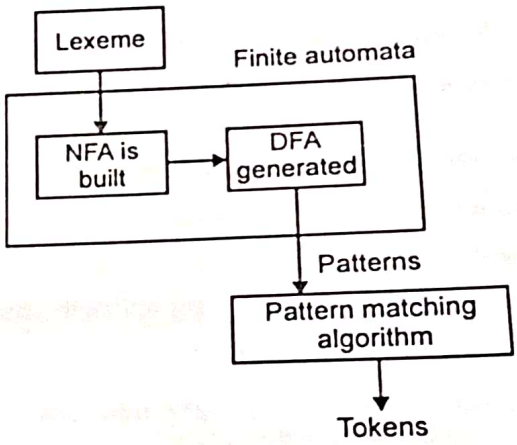


Fig. Role of finite automata

Q.46. Differentiate clearly between NFA and DFA.

OR Explain the point of differences between the determinism and non-determinism.

Ans.

Sr. No.	Non-deterministic finite automata (NFA)	Deterministic finite automata (DFA)
(1)	If the basic finite automata model is modified to allow zero, one, or more transitions from one state to another state on the same input symbol. The new finite automata is called NFA	DFA consist of a finite set of states and a set of transitions from state to state occur on single input symbols choosen from an alphabet Σ.
(2)	NFA is denoted by 5 tuple. $(Q, \Sigma, \delta, q_0, F)$ where, Q = finite set of states. Σ = set of input symbol. q_0 = start states. F = final states. δ = transition function from $Q \times \Sigma$ to Q.	DFA is denoted by 5 tuple $(Q, \Sigma, \delta, q_0, F)$ where, Q = set of states. Σ = set of input symbol. q_0 = start states. F = final states. δ = transition function from $Q \times \Sigma$ to Q.
(3)	In NFA there exists one or more transition from state to state on same input symbol.	In DFA, there exists exactly one transition from state to state on same input symbol.
(4)	In NFA state has an $\in$ - transition.	In DFA no state has an $\in$ - transition i.e. a transition on input $\in$.
(5)	NFA can recognize the regular set slower than DFA.	DFA can recognize the regular set faster than NFA.

Q.47. Give finite automata for a typical identifier of a programming language and also write the complete input required for LEX to produce an analyser that recognizes identifiers.

CS : S-09(5M), CT : S-09(6M)

Ans.

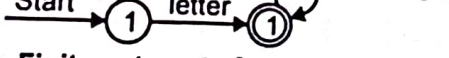


Fig. Finite automata for typical identifier

where, letter = a | b | ..... | Z | A | B | ..... | Z

digit = 0 | 1 | ..... | 19

Auxiliary definition :

letter = a | b | .....z | A | B | ..... | Z

digit = 0 | 1 | 2 |..... 19

identifier = letter (letter | digit) *

Translation rules :

identifier { LEXUAL = install id ( ) ; return (identifier) ; }

Lex program :

```
% { * Program for recognizing identifiers *}
% }
        letter [ A – Y a – z]
        digit [ 0 – 9]
        id letter (letter / digit / – ) *
        % %
        Auto / break / case / char / default / double /
        do / define / ..... { }
        {id} {printf ("\n identifier found % S", yy. Lex. 8);}
        % %
        main ( )
        {
                yy.lex ( ) ;
        }
```

Q.48. How is finite automata useful for lexical analysis? How tokens are generated in the following statement?

printf("%d%f%",rollno,per_mark); **CT : S-1**

Ans.

- A finite automata is an abstract machine that serves as an recog for the string that comprise a regular language.
- The idea is that we can feed an input string into a finite aut and will answer "yes" or "no" depending on whether or not the string belong to the language that the automata recognizes.
- Lexical analysis is the power of reading the source text of a pro and converting it into a sequence of tokens.
- The lexical structure of more or less every programming lang can be specified by a regular language, a common way to imple a lexical analyzer is to :

(1) Specify regular explanation for all of the kinds of token in the language. The disjunction of all the regular explanation thus describes any possible token in the language.

(2) Convert the overall regular expression specifying all possible token into a deterministic finite automata.

(3) Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.

- This approach is useful for programs called lexical analyzer generators exist to automate the entire processor.
- So in this way finite automata is useful for lexical analysis.

REGULAR EXPRESSIONS

Q.49. What is regular expression?

Ans. Regular expression :

- Regular expression are mathematical symbolisms which describe the set of strings of specific language.
- It provides convenient and useful notation for representing tokens.
- Here are some rules that describe definition of the regular expressions over the input set denoted by Σ.

(1) ε is a regular expression that denotes the set containing empty strings.

(2) If R1 and R2 are regular expression then R = R1 + R2 (same can also be represented as R = R1 | R2) is also regular expression which represents union operation.

(3) If R1 and R2 are regular expression then R = R1.R2 is also a regular expression which represents concatenation operation.

(4) If R1 is a regular expression than R = R1* is also a regular expression which represents kleen closure.

- A language denoted by regular expression is said to be a regular set or a regular language.

Example : Write a regular expression (R.E.) for a language containing the strings of length two over $\Sigma = (0,1)$

R.E. = (0 + 1) (0 + 1)

Q.50. Give the notations used for representing regular expression.

Ans. Notations used for representing regular expressions :

- Regular expression are tiny units, which are useful for representing the set of strings belonging to some specific language.
- Consider notations used for writing the regular expressions.

(1) **One or more instances :**

- To represent one or more instances + sign is used. If r is a regular expression then r* denotes one or more occurrences of r.
- For example, set of strings in which there are one or more occurrences of 'a' over the input set (a) then the regular expression can be written as a*. It basically denotes the set of {a, aa, aaa, aaaa, ...}

(2) **Zero or more instances :**

- To represent zero or more instances * sign is used. If r is a regular expression then r* denotes zero or more occurrences of r.
- For example, set of strings in which there are zero or more occurrences of 'a' over the input set (a) then the regular expression can be written as a*. It basically denotes the set of { ε, a, aa, aaa, ...}.

(3) **Character class :**

- A class of symbols can be denoted by [ ].
- For example, [012] means 0 or 1 or 2. Similarly a complete class of a small letters from a to z can be represented by a regular expression [a-z]. The hyphen indicates the range. We can also write a regular expression for representing any word of small letters as [a-z]*.

Q.51. List the rules for constructing regular expressions. Write some properties to compose additional regular expressions. What is a regular definition? Give a suitable example.

Ans.

- The rules for constructing regular expressions over some alphabet Σ are divided into two major classifications which are as follows :

(i) Basic rules (ii) Induction rules

(i) **Basic rules :** There are two rules that form the basis :

(1) ϵ is a regular expression, and L {ϵ} is {ϵ}, that is its language contains only an empty string.

(2) If a is a symbol in Σ, then a is a regular expression, and L {a} = {a}, which implies the language with one string, of length one, with a in its one position.

(ii) **Induction rules :**

- There are four induction rules that built larger regular expression recursively from smaller regular expressions.
- Suppose R and S are regular expression with languages L {R} and L {S}, respectively.

(1) (R) (S) is a regular expression representing the language L (R) . L (S).

(2) (R)/(S) is a regular expression representing the language L (R) ∪ L (S)

(3) $(R)^*$ is a regular expression representing the language $(L (R))^*$

(4) (R) is a regular expression representing L (R). This rule states that additional pairs of parentheses can be added around expressions without modifying the language.

Properties of regular expression :

- To compose additional regular expressions, the following properties are to be considered, a finite number of times :

(1) If a_1 is a regular expression, then (a_1) is also a regular expression.

(2) If a_1 is a regular expression, then a_1' is also a regular expression.

(3) If a_1 and a_2 are two regular expressions, then $a_1 a_2$ is also a regular expression.

(4) If a_1 and a_2 are two regular expressions, then $a_1 + a_2$ is also a regular expression.

Regular definition : If Σ = alphabet set, then a regular definition is a sequence of definitions of the form :

$D_1 \rightarrow R_1$

$D_2 \rightarrow R_2$

.....

$D_n \rightarrow R_n$

where,

D_2 is a new symbol, not in Σ and not the same as any of the other D's.

R_1 is a regular expression over the alphabet $\Sigma \cup (D_1, D_2, ...D_{1-1})$

For example, let us consider the C identifiers that are strings of letters, digits, and underscores. Here, we give a regular definition for the language of C identifiers.

letter_ → A | B | .... | Z | a | b | ..... | z | _

digit → 0 | 1 | ..... | 9.

id → letter_ (letter_ | digit)*

APPENDIX

UNIT - I

JUST-IN-TIME COMPILERS

Q.1. Explain Just-In-Time Compilers.

Ans. Just-In-Time (JIT) Compiler:

- Just-In-Time (JIT) compiler is a program that turns bytecode into instructions that can be sent directly to a computer's processor (CPU). Typically, compilers are key in deciding the speed of an application for developers and end users. Just-in-time compilers can be used for performance optimization to improve application runtime.
- As opposed to other compiler types, the main defining characteristic of a JIT compiler is that a JIT compiler runs after a program starts and compiles code. A common way to say this is that a JIT compiler compiles code on the fly, or in other words, just in time.
- The compiled code, or bytecode sequences, are turned into a faster, more readable machine language, or native code. Translating source code into a language that a host CPU natively understands typically means a faster, more easily readable instruction set.
- JIT compilers contrast different compiler types such as a traditional compiler, which will compile all code to a machine language before a program starts to run. Newer programs will make use of JIT compilers, which generate code while the program is running. Two common uses of JIT compilers include Java Virtual Machine (JVM) which is used in Java, as well as CLR (Common Language Runtime) which is used in C#.
  For example, in the Java programming language and environment, a just-in-time (JIT) compiler turns Java bytecode -- a program that contains instructions that must be interpreted -- into instructions that can be sent directly to the processor.
- JIT compilers are utilized commonly in a Java Runtime Environment, as they are used to optimize performance in java-based applications.

Q.2. How does Just-in-Time Compilation work?

Ans. JIT Compilation:

- Bytecode is an important feature in applications that help in cross-platform execution. Additionally, bytecode must be compiled and translated to a language a CPU can properly understand. However, how that bytecode is translated into a native language may have a large impact on the speed and performance of an application.
- To improve performance, JIT compilers will, at runtime, compile suitable bytecode sequences into machine code. The code is typically sent to a processor, where the code instructions are then carried out.
- When the same block of bytecode is needed again, the previously created object code will be used. Code that looks like it can be re-optimized is called "hot." Code can be monitored, and hot code paths can be created to optimize code, as opposed to having the same sequence of code be interpreted multiple times -- which may occur in other compiler types.
- With less chance of code being interpreted multiple times, there is less overhead, meaning faster execution speeds. This is why most implementations of JVM use JIT compilers.

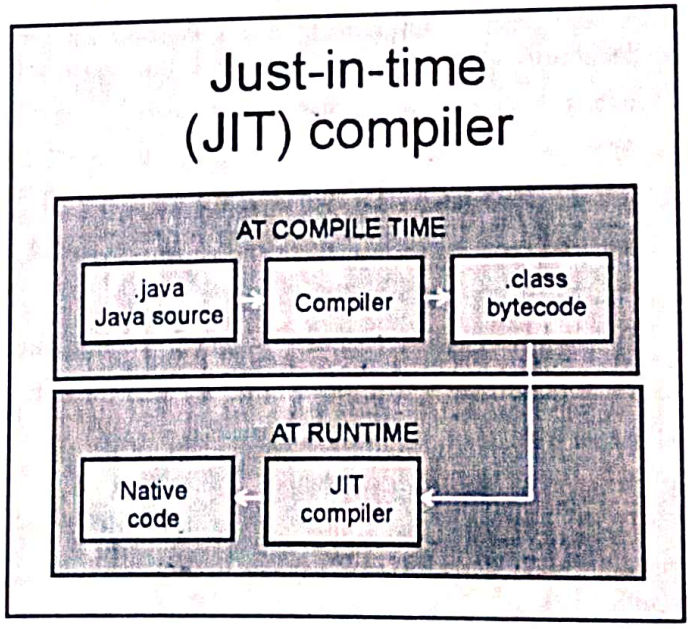


Fig. A visual of how a just-in-time (JIT) compiler works.

- A JIT compiler can also make relatively simple optimizations when compiling bytecode into a native machine language. As an example, a JIT compiler can get rid of common sub-expressions, reduce memory access in register allocations, and perform data-analysis and register operations by translating from stack operations. However, because of the time it takes to load and compile bytecode, there is a startup delay in the initial execution of an application.
- To help anticipate startup times, a good rule of thumb to follow is that the more JIT compilers are used to optimize a system, the longer the initial startup delay will be.
- There are a few ways to decrease initial startup delays, such as separating startup modes. For example, including both a client and server mode could allow minimal compilation and optimization when in one mode versus the other, meaning that the chosen mode will have a faster startup time. Another way is to combine JIT compilers with either AOT (ahead-of-time) compilers or interpreters.

Phases of Just-in-Time Compilation:

- As mentioned, a JIT compiler will compile suitable bytecode sequences into machine code, and then the code is sent to a processor where the code instructions are carried out.
- JIT compilation can also be separated by different levels of optimization: cold, warm, hot, very hot and scorching. Each optimization level is set to provide a certain level of performance. The initial, or default, level of optimization is called warm, while code that looks like it can be further re-optimized is called hot.
- This level increases upwards until scorching, with each level being of higher importance pertaining to performance that can be re-optimized. Through sampling, a JIT compiler can determine which methods appear more often at the top of a stack. The optimization level can also be downgraded to cold, to further improve startup time.

Q.3. Compare JIT, Interpreters and Ahead-of-Time Compilers.

Ans. Just-in-Time Compilers vs. Interpreters vs. Ahead-of-time Compilers:

- In general, interpreters and compilers can both be used to translate machine language. An interpreter will commonly perform tasks such as Parsing, type checking and lexing, and does the same kind of work as a compiler.
- Interpreters have a fast startup time, don't have to go through a compilation step and will translate code one line at a time, on the fly. However, an interpreter may be slower than a compiler in a case where an application runs the same code multiple times -- since the same translation must happen as many times as the code is repeated.
- A typical compiler does not translate on the fly, as it translates compiled code before the application begins running. Because of this, its startup time is a bit longer, but any code loops that appear will run faster since the code translation does not need to be repeated multiple times. Compilers have more time to look at the code and make any optimizations as well. JIT compilers combine both principals to get the best of both worlds.
- Ahead-of-time or AOT compilers, compile code into a native machine language similar to a normal compiler; however, AOT will transform bytecode of a virtual machine into machine code.

Advantages of Just-in-Time Compilation:

Advantages of JIT compilation include:

(i) JIT compilers need less memory usage.
(ii) JIT compilers run after a program starts.
(iii) Code optimization can be done while the code is running.
(iv) Any page faults can be reduced.
(v) Code that is used together will be localized on the same page.
(vi) Can utilize different levels of optimization.

Disadvantages of Just-in-Time Compilation:

Disadvantages of JIT compilation include:

(i) Startup time can take a noticeable amount of time.

(ii) Heavy usage of cache memory.

(iii) Increases the level of complexity in a Java program.

UNIT - II

PREDICTIVE PARSER

Q.4. Explain Predictive Parser Algorithm with example.

Ans. Predictive Parser:

A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

Consider

A -> A1 | A2 | ... | An

If the non-terminal is to be further expanded to 'A', the rule is selected based on the current input symbol 'a' only.

Predictive Parser Algorithm:

(1) Make a transition diagram (DFA/NFA) for every rule of grammar.

(2) Optimize the DFA by reducing the number of states, yielding the final transition diagram.

(3) Simulate the string on the transition diagram to parse a string.

(4) If the transition diagram reaches an accept state after the input is consumed, it is parsed.

Consider the following grammar:

E->E+T|T

T->T*F|F

F->(E)|id

After removing left recursion, left factoring

E->TT'

T'->+TT'|ε

T->FT"

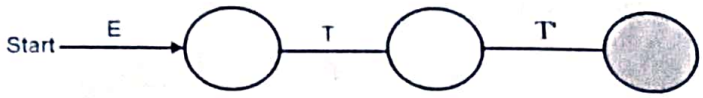
T"->*FT"|ε

F->(E)|id

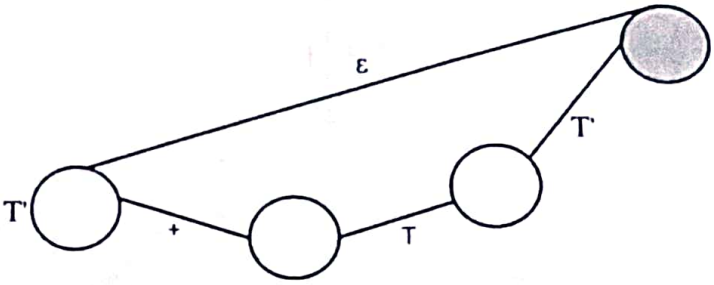
STEP 1:

Make a transition diagram(DFA/NFA) for every rule of grammar.

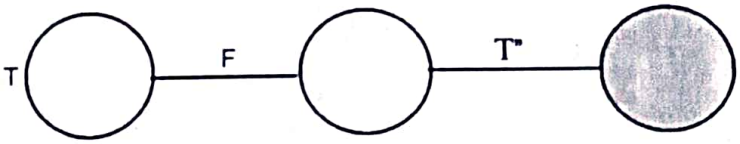
- E->TT'



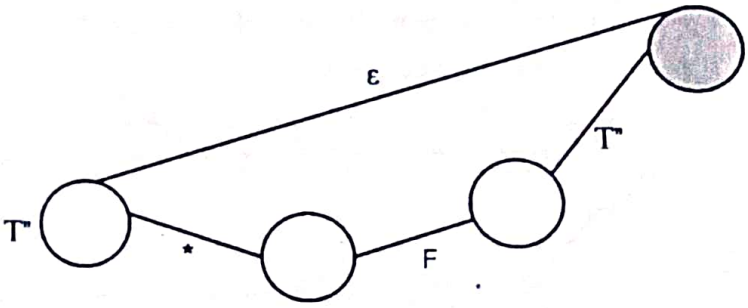
- T'->+TT'|ε



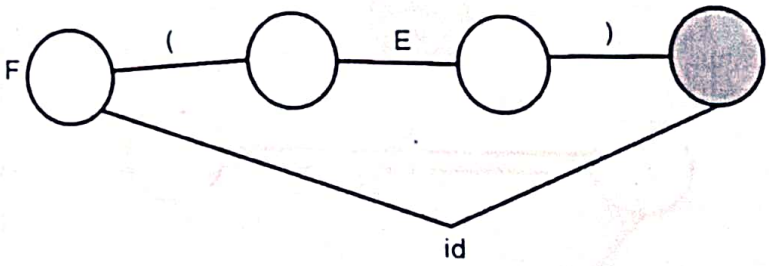
T->FT"



- T"->*FT"|ε



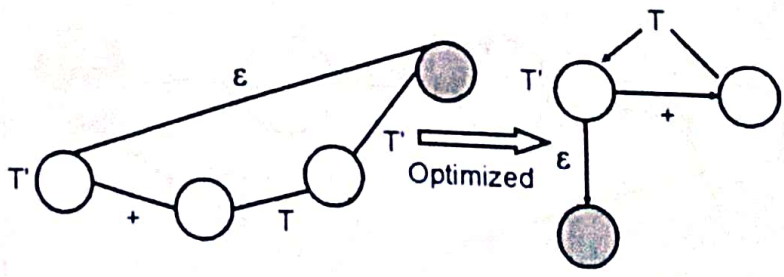
- F->(E)|id



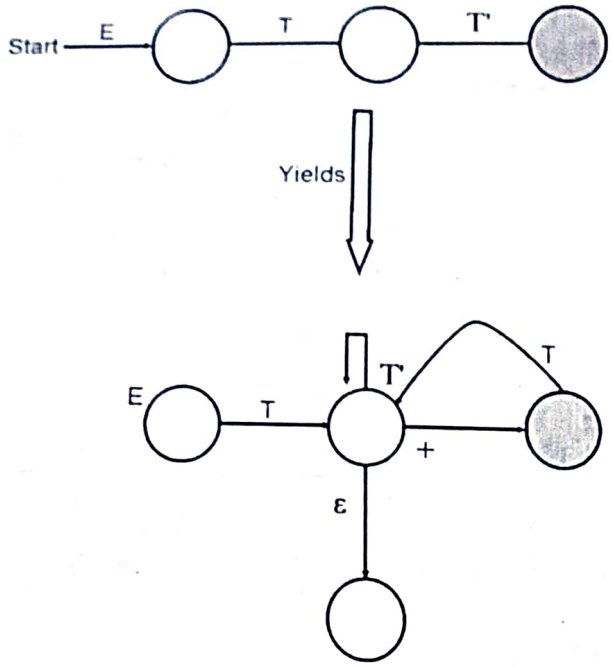
STEP 2:

Optimize the DFA by decreases the number of states, yielding the final transition diagram.

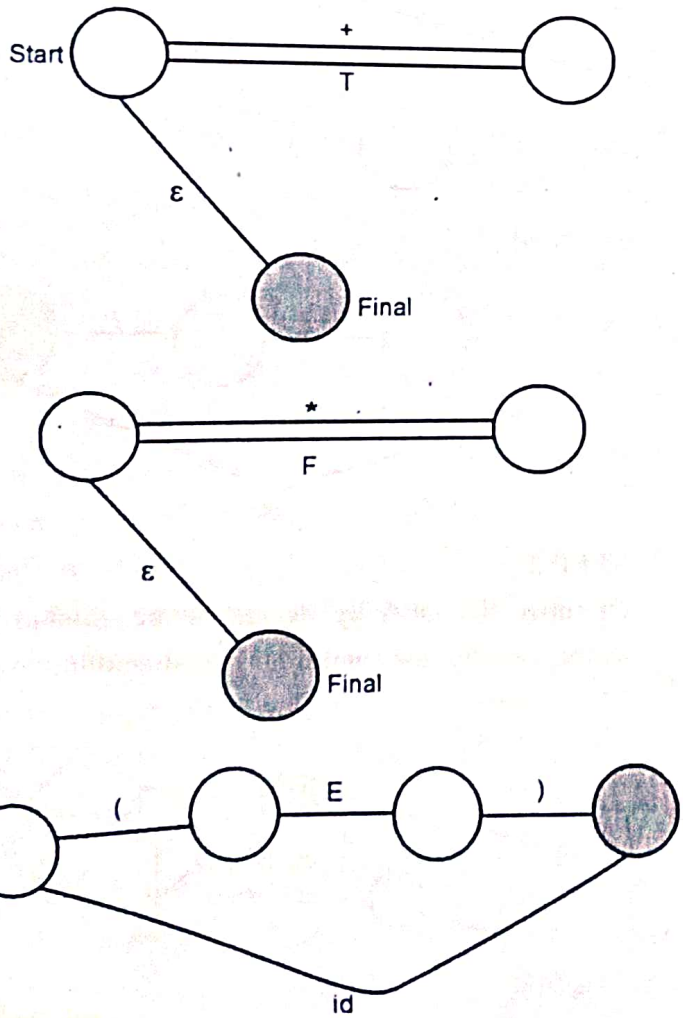
T'->+TT'|ε



It can be optimized ahead by combining it with DFA for E->TT'



Accordingly, we optimize the other structures to produce the following DFA



STEP 3:

Simulation on the input string:

Steps involved in the simulation procedure are:

(1) Start from the starting state.

(2) If a terminal arrives consume it, move to the next state.

(3) If a non-terminal arrive go to the state of the DFA of the non-terminal and return on reached up to the final state.

(4) Return to actual DFA and Keep doing parsing.

(5) If one completes reading the input string completely, you reach a final state, and the string is successfully parsed.

APPENDIX - II

UNIT - I

RELATING REGULAR EXPRESSION AND FINITE AUTOMATA

Q.1. Explain the relationship between Finite Automata and Regular Expression.

Ans. Regular Expression (RE):

Regular expression is the language which is used to describe the language and is accepted by finite automata. Regular expressions are the most effective way to represent any language. Let Σ be an alphabet which denotes the input set.

The regular expression over Σ can be defined as follows:

(i) Φ is a regular expression which denotes the empty set.

(ii) ε is a regular expression and denotes the set {ε} and it is called a null string.

(iii) For each 'a' in Σ 'a' is a regular expression and denotes the set {a}.

(iv) If r and s are regular expressions denoting the language. L_1 and L_2 respectively then,

- r + s is equivalent to $L_1 \cup L_2$ union
- rs is equivalent to $L_1 L_2$ concatenation
- r* is equivalent to L_1* closure

The r* is known as Kleen closure or closure which indicates occurrence of r for an infinite number of times.

Finite Automata (FA):

Finite automata is an abstract computing device. It is a mathematical model of a system with discrete inputs, outputs, states and a set of transitions from state to state that occurs on input symbols from the alphabet Σ.

Finite automata is defined as a 5-tuples

$M = (Q, \Sigma, \delta, q_0, F)$

Where,

Q: Finite set called states.

Σ: Finite set called alphabets.

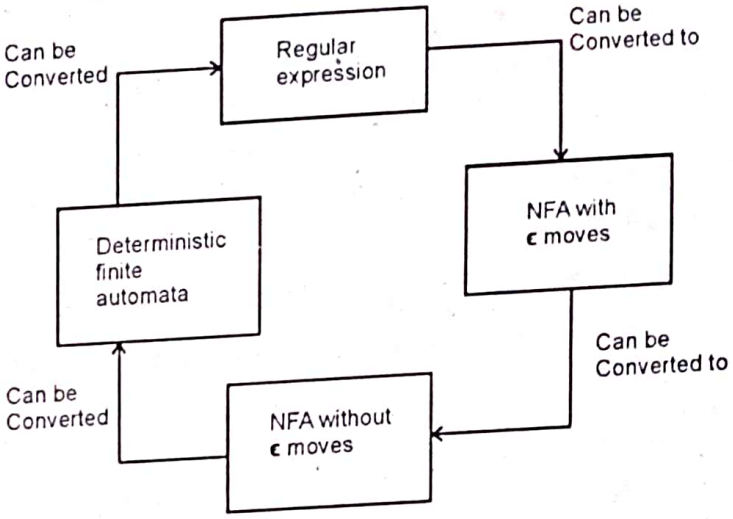
$\delta: Q \times \Sigma \rightarrow Q$ is the transition function.

$q_0 \in Q$ is the start or initial state.

F: Final or accept state.

Relationship:

The relationship between FA and RE is as follows:

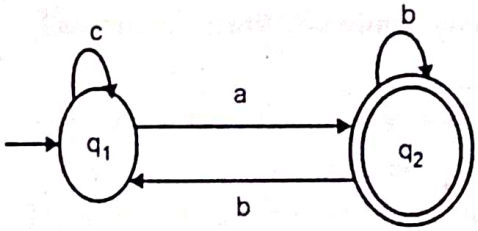


The above figure explains that it is easy to convert

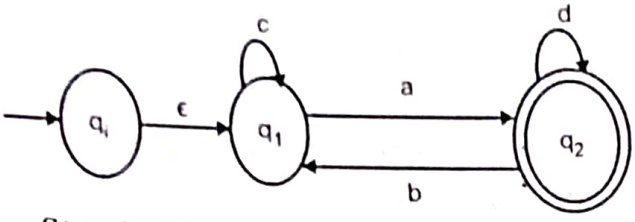
- RE to Non-deterministic finite automata (NFA) with epsilon moves.
- NFA with epsilon moves to without epsilon moves.
- NFA without epsilon moves to Deterministic Finite Automata (DFA).
- DFA can be converted easily to RE.

Q.2. Generate a Regular Expression for a Finite Automata with state elimination method.

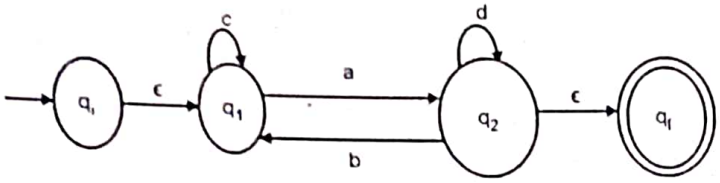
Ans. Conversion of Finite Automata into equivalent Regular Expression using the State Elimination Method is explained below:



Step 1: Initial state q_1 has incoming edge. So, create a new initial state q_i.



Step 2: Final state q_2 has outgoing edge. So, create a new final state.



Step 3: Start eliminating intermediate states one after another.

If there is path going to q_i to q_2 via q_1

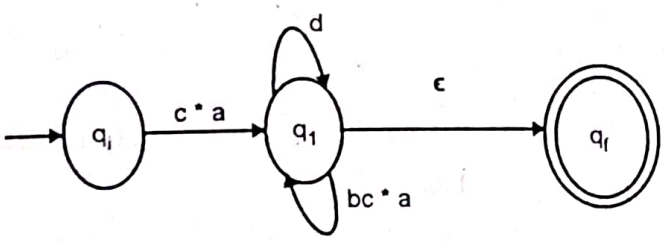
So after eliminating q_1 we can connect a direct path from q_i to q_2 having cost

$\varepsilon .c^*.a = c^*a$

There is a loop on q_2 using state q_i

So after eliminating state q_1 we can draw a direct loop on state q_2 having cost

$b.c^*.a = b.c^*.a$



Step 4: Now eliminate q_2.

After eliminating q_2 direct path from state q_i to q_f having cost.

$c^*a(d + bc^*a)^* \varepsilon = c^*a(d + bc^*a)^*$

Q.3. Can we convert non-deterministic finite automata into deterministic finite Automata?

Ans.

- Yes, we can convert a NFA into DFA. For every NFA there exists an equivalent DFA. The equivalence is defined in terms of languages acceptance. Since NFA is nothing but a finite automata in which zero, one or more transitions on an input symbols are permitted, it can always construct finite automata which will simulate all moves of DFA on a particular input symbol in parallel, then get a finite automata in which there will be exactly one transition on every input symbol. Here, corresponding to a NFA there exist a DFA.
- To construct DFA equivalent to NFA, it should be remembered that states of DFA are a collection of states of NFA.

Algorithm NFA - to - DFA:

Input: NFA with set of states

$N = \{n_0, n_1 ..... n_n\}$, with start state n_0.

Output: DFA, with set of states $D' = \{d_0, d_1, d_2 ..... d_n\}$, with start state d_0.

$d_0 = \varepsilon - \text{closure}(n_0)$

$D' = \{d_0\}$

set d_0 unmarked

while there is an unmarked state d in D'. {

set d marked {

For each input symbol 'a' {

Let T be a set of states in NFA to which there is a transition on 'a' from some state n_i in d d' = ε-closure (T).

If d' is not already present in D' {

D'=D' ∪ {d'}

Add transition d⟶d',labeled 'a'

set d' unmarked

}

}

}

}

Q.4. Design Lexical Analyzer for the following LEX Program:

AUXILIARY DEFINITIONS

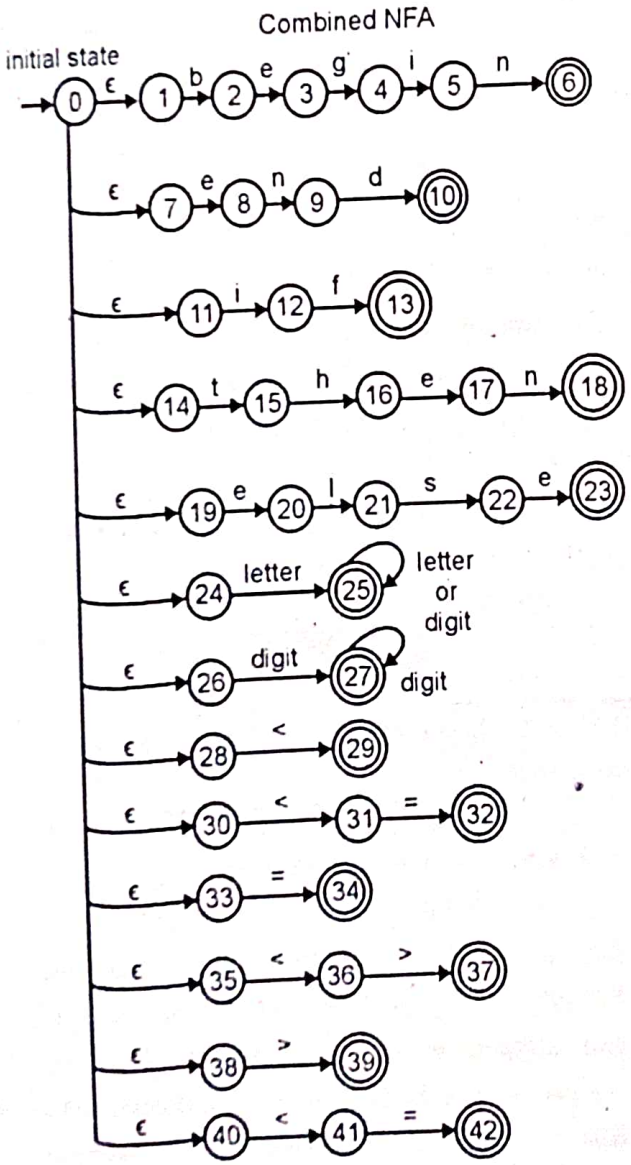
letter = A|B|C|.....|Z

digit = 0 |1|2|.....|9

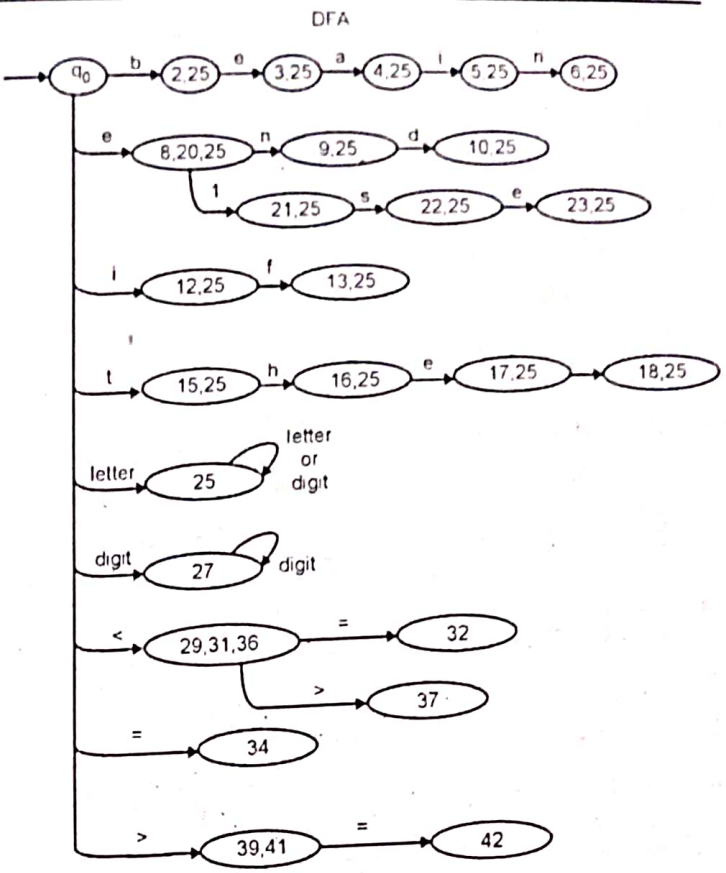
TRANSLATION RULES

begin {return 1}
end {return 2}
If {return 3}
then {return 4}
else {return 5}
letter (letter+digit)* {value=Install ( );return 6}
digit + {value=Install ( );return 7}
< {value=1;return 8}
<= {value=2;return 8}
= {value=3;return 8}
<> {value=4;return 8}
> {value=5;return 8}
>= {value=6;return 8}

Ans. The combined NFA for various patterns will be



Convert NFA to DFA – The corresponding DFA will be



States {0, 1, 7, 11, 14, 19, 24, 26, 28, 30, 33, 35, 38, 40} are combined and named as q_0 to make a starting state of DFA.

In combined NFA, transitions from state 1 to 2 and from state 24 to 25 are the same because Input 'b' is a letter.

States 2, 25 are combined. Similarly, other states are combined.

States 29, 31 and 36 are combined because they all reach after getting the input '<'.

Similarly, other states are combined.

UNIT - II

PUSHDOWN AUTOMATA

Q.5. Explain Pushdown Automata.

Ans. Pushdown Automata:

- A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

A pushdown automaton is:
"Finite state machine" + "a stack"

- A pushdown automaton has three components:

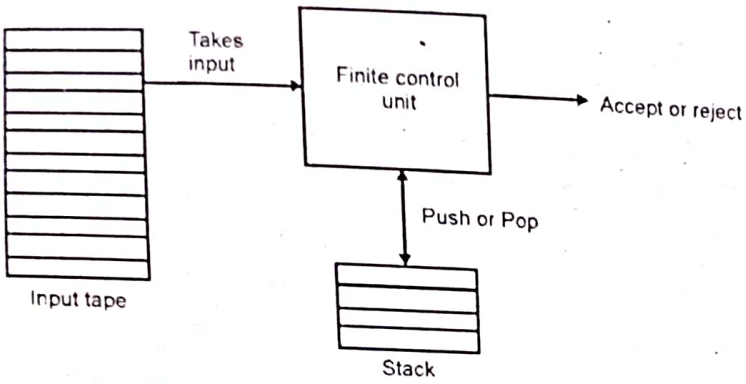
(i) An input tape.
(ii) A control unit, and
(iii) A stack with infinite size.

The stack head scans the top symbol of the stack.

- A stack does two operations:

(i) **Push:** A new symbol is added at the top.
(ii) **Pop:** The top symbol is read and removed.

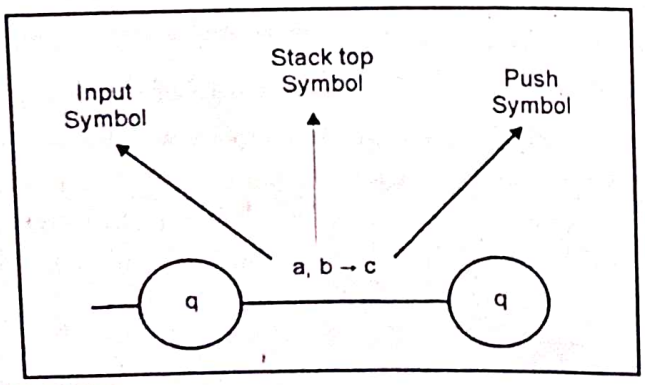
- A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



- A PDA can be formally described as a 7-tuple $(Q, \sum, S, \delta, q_0, I, F)$:

where
Q - The finite number of states
$\sum$ - Input alphabet
S - Stack symbols
δ - The transition function:
$Q \times (\sum \cup \{\varepsilon\}) \times S \times Q \times S^*$
q_0 - The initial state $(q_0 \in Q)$
I - The initial stack top symbol $(I \in S)$
F - A set of accepting states $(F \in Q)$

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as $a,b \rightarrow c$:



This means at state q_1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q_2.

Terminologies Related to PDA:

(i) **Instantaneous Description:**

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where
q - is the state
w - is unconsumed input
s - is the stack contents

(ii) **Turnstile Notation:**

- The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol " $\vdash$ ".
- Consider a PDA $(Q, \sum, S, \delta, q_0, I, F)$. A transition can be mathematically represented by the following turnstile notation:
  $(p, aw, T\beta) \vdash (q, w, \alpha b)$
- This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed and the top of the stack 'T' is replaced by a new string 'α'.

Note: If we want zero or more moves of a PDA, we have to use the symbol ($\vdash$ *) for it.

UNIT - III

SEMANTIC ANALYSIS: ATTRIBUTE GRAMMER

Q.6. Explain Semantic Analysis.

Ans. Semantic Analysis:

- Semantic analysis is the third phase of Compiler. Semantic analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

- Type checking is an important part of semantic analysis where compiler makes sure that each operator has matching operands.

Semantic Analyzer:

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition. It gathers type information and stores it in either syntax tree or symbol table. This type information is subsequently used by compiler during intermediate-code generation.

Semantic Errors:

Errors recognized by semantic analyzer are as follows:

(a) Type mismatch
(b) Undeclared variables
(c) Reserved identifier misuse

Functions of Semantic Analysis:

(1) **Type Checking:**
Ensures that data types are used in a way consistent with their definition.

(2) **Label Checking:**
A program should contain labels references.

(3) **Flow Control Check:**
Keeps a check that control structures are used in a proper manner. (example: no break statement outside a loop).

Example:

```
float x = 10.1;
float y = x*30;
```

In the above example integer 30 will be type casted to float 30.0 before multiplication, by semantic analyzer.

Static and Dynamic Semantics:

(1) **Static Semantics:**
It is named so because of the fact that these are checked at compile time. The static semantics and meaning of program during execution, are indirectly related.

(2) **Dynamic Semantic Analysis:**
It defines the meaning of different units of program like expressions and statements. These are checked at runtime unlike static semantics.

RUN-TIME ENVIRONMENT, PROCEDURE ACTIVATION

Q.7. Explain Run-Time Environment.

Ans. Run-Time Environment:

- A Program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.
- By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.
- Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Q.8. Explain Activation Trees.

OR Explain Procedure Activation.

Ans. Activation Trees:

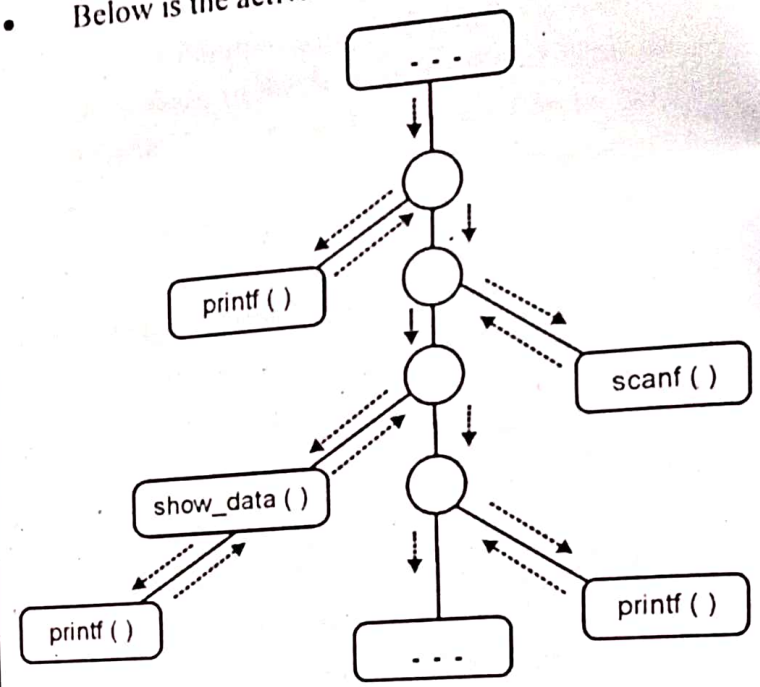
- A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.
- The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e. parameters which are used to send input to the called procedure.
Return Value	Stores return values.

- Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.
- We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the activation tree.
- To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue...");
. . .
int show_data(char *user)
{
    printf("Your name is %s", username);
    return 0;
}
. . .
```

- Below is the activation tree of the code given.



- The procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

PARAMETER PASSING, VALUE RETURN

Q.9. Explain Parameter Passing.

Ans. Parameter Passing:

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

(i) r-value:

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

(ii) l-value:

- The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

- From this example, we understand that constant values like 1, 7, 12 and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

(1) Formal Parameters:

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

(2) Actual Parameters:

- Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
}
fun_two(int formal_parameter)
{
    print formal_parameter;
}
```

- Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

(a) Pass by Value:

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

(b) Pass by Reference:

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

(c) Pass by Copy-restore:

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

(d) Pass by Name:

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

MEMORY ALLOCATION STORAGE ALLOCATION

Q.10. Explain Storage Allocation.

OR Explain Storage Allocation. What are the different Storage Allocations?

Ans. Storage Allocation:

Runtime environment manages runtime memory requirements for the following entities:

(i) Code:

It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

(ii) Procedures:

Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.

(iii) Variables:

Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Types of Allocation:

(1) Static Allocation

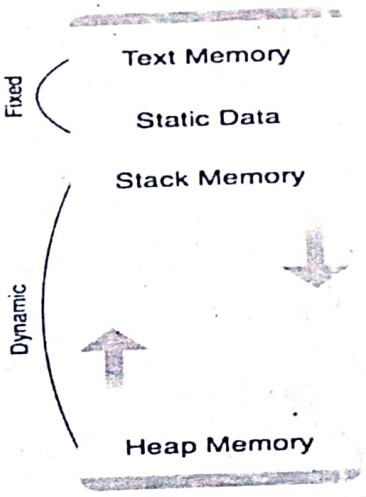
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

(2) Stack Allocation:

Procedure calls and their activations are managed by means of stack memory allocation. It works in Last-In-First-Out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

(3) Heap Allocation

- Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



- As shown in the above Fig. the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Q.11. Explain Memory allocation.

OR What are the techniques of storage allocation in compiler design?

Ans. There are various storage allocation techniques which are as follows: