CD-UNIT-1

1 Language Processing System :

Language processing system is a system that translates the source language which is taken as input into machine language. This translation can be done by dividing the source file into modules. These modules are as follows,

1. Preprocessor
2. Compiler
3. Assembler
4. Linker/Loader.

The typical language processing system is shown in the figure below.

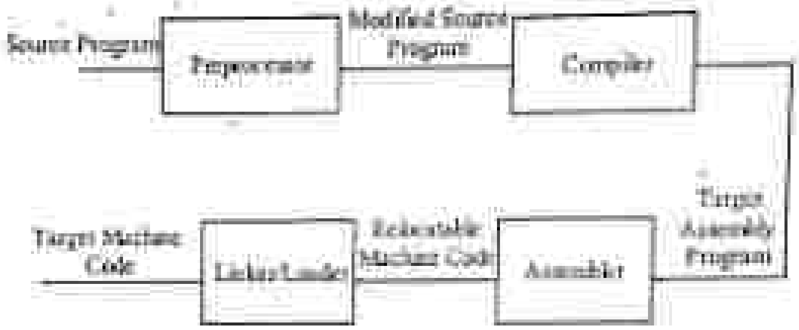


Figure: Language-Processing System

1. Preprocessor

It is a special program processing the code prior to the actual translation to perform necessary functions like deleting comments, adding necessary files and doing macro substitutions.

2. Compiler

A compiler is a program that converts a sources program into a target program.

3. Assembler

Assembler is a translator which translates assembly language program into object code. This program specifies symbolic form of the machine language of the computer.

4. Linker and Loader

❖ **Linkers**

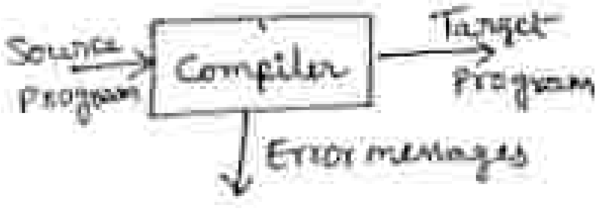
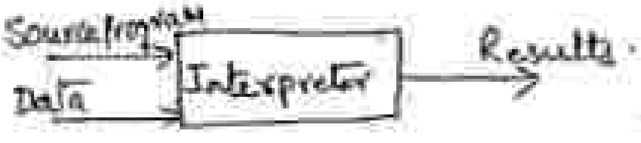
It is a program that links the two object files containing the compiled or assembled code to form a single file which can be directly executable. It is also responsible for performing the following functions,

1. Linking the object program with that of the code for standard library functions and
2. Resources provided by the operating system like memory allocators and input and output devices.

❖ **Loaders**

It is a program which loads or resolves all the code (relocatable code) whose principal memory references have undetermined initial locations present anywhere in the memory. It resolves with respect to the given base or initial address.

(4) Differences between Compiler and Interpreter :-

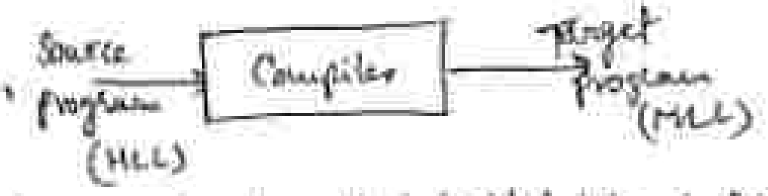
compiler	Interpreter
① compiler is a program used to convert highlevel language to machine code and it executes whole program at a time	① It is same like compiler, but it executes only one statement at a time
② Some compilation languages are C, C++, FORTRAN, PASCAL, Ada etc	② Some Interpretation languages are LISP, ML, Prolog, Smalltalk etc.
③ The translation process carried out by a compiler is termed as compilation	③ The translation process carried out by the Interpreter is termed as Interpretation.
④ Processing time is less	④ Processing time is more
⑤ compilers are large in size and occupy more memory	⑤ Interpreters are smaller than compilers.
⑥ The object program execution is not carried out by the compiler	⑥ The object program execution is carried out by the Compiler Inter
⑦ It process each program statement exactly once	⑦ It might process some statements repeatedly.
⑧ The execution speed is more in a compiler.	⑧ The execution speed is less in an Interpreter.
⑨ The compiler diagram is  Source program Compiler Target Program Error messages	⑨ The Interpreter diagram is 
⑩ The cost of decoding must be paid at once after the entire program is executed	⑩ the cost of decoding must be paid each time the statement is to be executed.
⑪ It is also called as Software translation	⑪ It is also called as Software simulation.

② Structure of a compiler :

→ A compiler is a program that reads a program written in one language and translates it into an equivalent program in another language and the compiler reports to its users, the presence of errors in the source program.

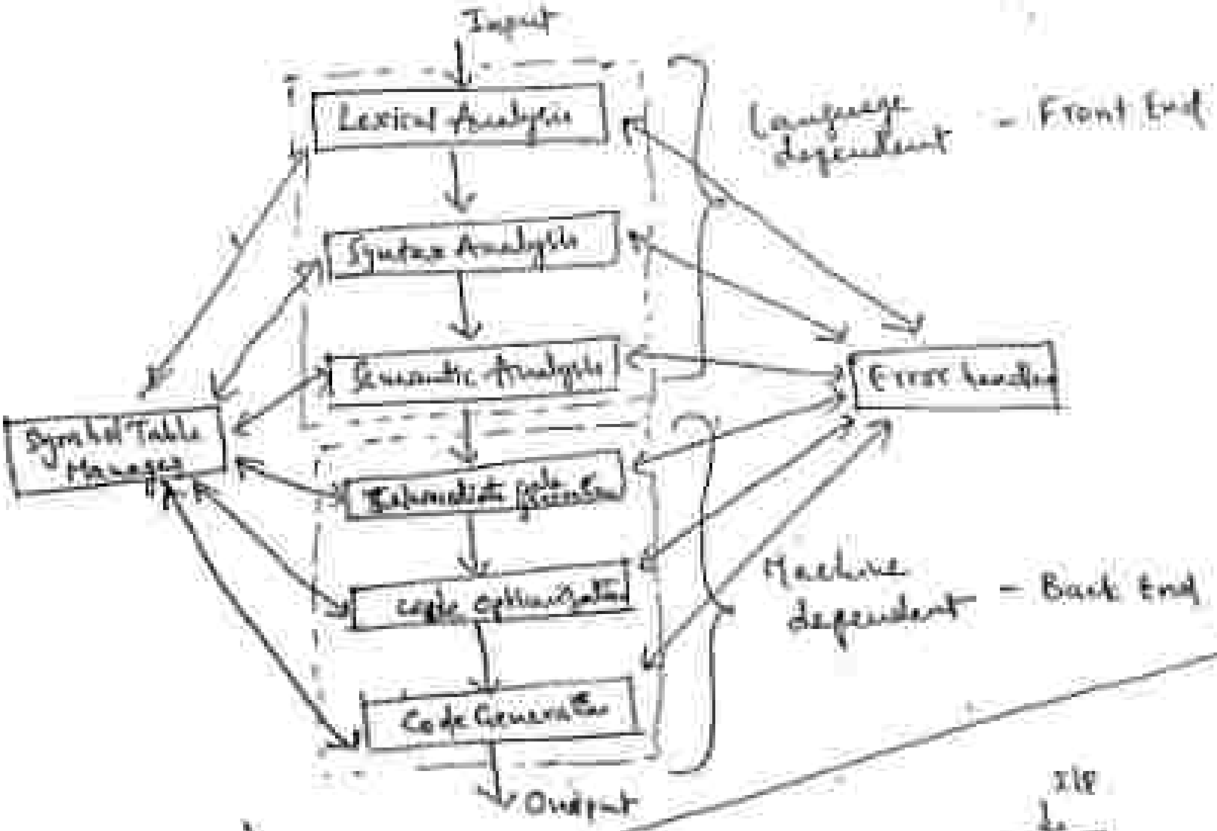
→ The program written is called source program.

→ The translated program is called as target program.

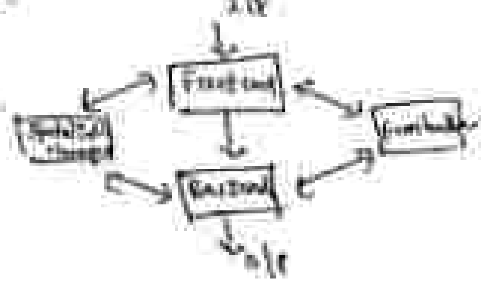


→ A compiler can be broadly divided into 2 parts based on whether the phase is

- Language dependent
- Machine dependent



(or)



phases of a compiler

→ A compiler is divided into number of parts, segments (or) sections and each section is known as a phase.

→ A compiler divided into two parts

- Analysis part (Front end)
- Synthesis part (Back end)

→ The Analysis part is machine Independent and Language dependent

→ The Sythesis part is machine dependent and Language Independent.

→ The Analysis part is divided into three phases

a) Lexical Analyzer / Scanner

b) Syntax Analyzer / Parser

c) Semantic Analyzer.

→ The Synthesis part is divided into three phases

d) Intermediate code Generator

e) Code optimizer.

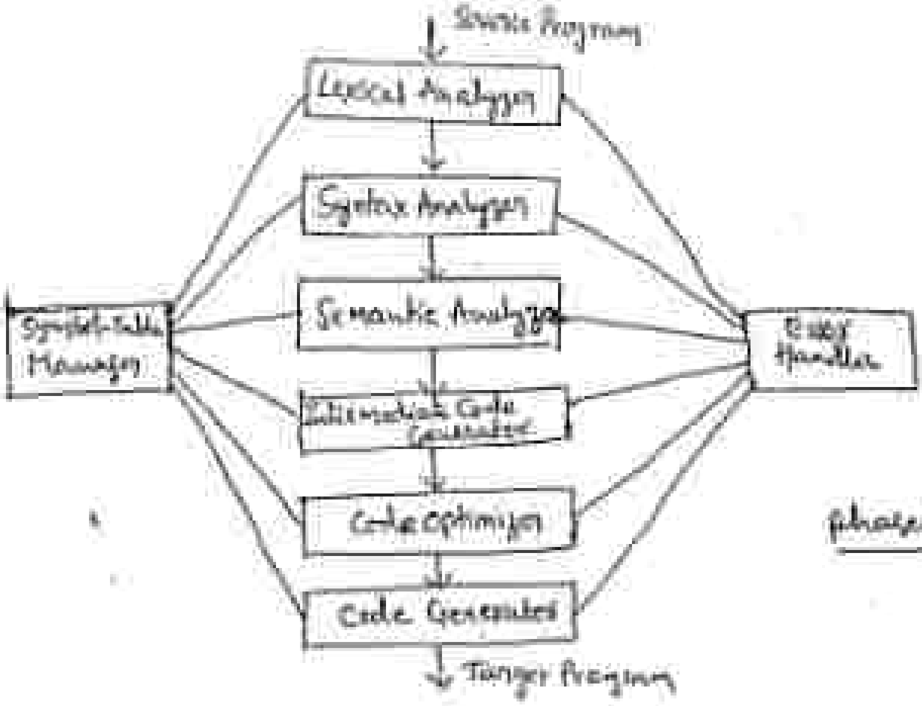
f) Code Generator

→ Two Additional activities of a compiler are,

g) Symbol table Manager

h) Error handler

→ The phases of a Compiler is depicted as,



phases of a compiler

Lexical Analyzer: (Scanner)

→ The lexical Analyzer does lexical Analysis.

→ This phase reads the characters in the source program and groups them into a stream of tokens.

→ Tokens are the sequence of characters having a collective meaning.

→ The character sequence forming a token is called the lexeme.

Example:

Consider an Assignment Statement

position := initial + rate * 60.

lexeme	Token
position	id
:=	Assignment Symbol
initial	id
+	plus sign
rate	id
*	Multiplication
60	Numeric Literal

→ The statement after the lexical Analysis is given by,

$$\boxed{id_1 := id_2 + id_3 * 60}$$

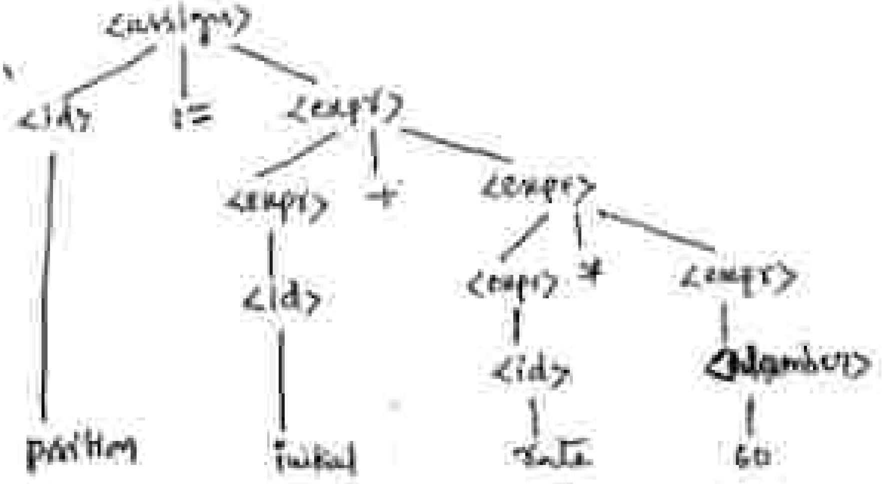
- where id_1, id_2, id_3 are tokens for position, initial and rate respectively.

b) Syntax Analyzer: (Parser)

→ The Syntax Analyzer does Syntax Analysis.

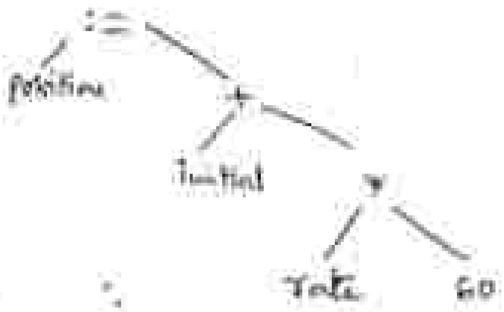
→ It groups the tokens into grammatical phrases represented by a hierarchical structure called a parse tree.

→ The parse tree for the input string $id_1 := id_2 + id_3 * 60$ is,



→ It can be also represented by a syntax tree, which is a compressed representation of the parse tree.

→ The Syntax tree is,



(c) Semantic Analyzer:

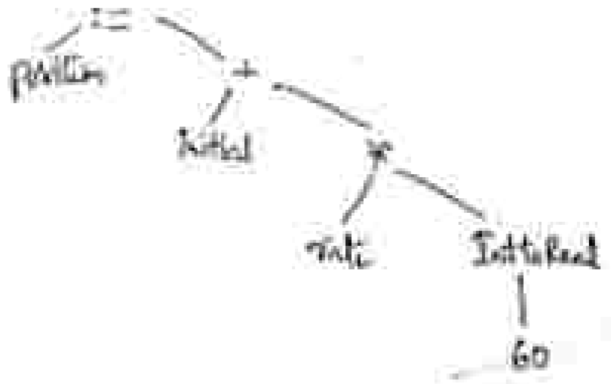
→ The Semantic Analyzer does the Semantic Analysis

→ Semantic Analysis checks the source program for Semantic errors.

→ It performs type checking.

Example:

- For the input string, $id_1 := id_2 + id_3 * 60$
  Let all identifiers are real, then we have to convert the number 60 as Inttoreal.
- The abstract Syntax tree after Semantic Analysis is,



d) Intermediate Code Generator:

→ After Syntax and Semantic Analysis, the compiler generates an intermediate representation of the source program.

→ One popular type is "Three Address Code".

Example

→ The three Address code for the statement
position := Initial + rate * 60 is given as,

```
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
 id1 := temp3
```

e) ~~Code Optimizer~~

~~→ Intermediate~~

e) Code optimizer:

→ The code optimization phase improves intermediate code

→ The code $id_1 := id_2 + id_3 * 60$ can be optimized as,

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

f) Code Generation:

→ The code generation phase converts the intermediate code into a target code.

→ The target code consisting of sequenced machine code (or) assembly code that perform the same task.

Example: Assembly code for $id_1 := id_2 + id_3 * 60$ is,

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

→ The # signifies that 60.0 is treated as constant.

g) Symbol Table Manager:

→ A symbol table management (or) bookkeeping is a portion of the compiler which keeps track of the names used by the program and records information such as datatype, precision, and so on.

→ The data structure used to record this information is called Symbol table.

→ This datastructure allows us to find the record for each identifier quickly and to store (or) retrieve data quickly.

h) Error handler:

→ Error handler is invoked when a fault in the source program is detected.

→ The Syntax and Semantic Analysis phases usually handle a large number of errors.

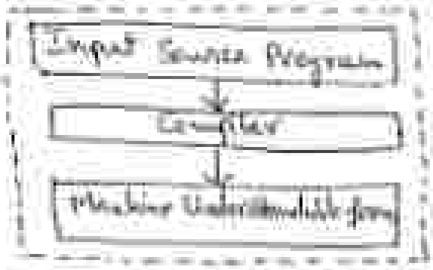
Example: Errors detected in different phases

lexical Analysis	—	Token misspelled
Syntax Analysis	—	missing parenthesis (or) semicolon
Semantic Analysis	—	incompatible types
Intermediate CG	—	statements never reached
, Code generator	—	~~to~~ large to fit in a word.

(*) Pass of a compiler.

→ A pass is one complete scan of the program, which includes reading an input source program and converting the source program into machine readable form.

→ The logical structure of a pass :-



→ Now, any compiler cannot be designed as a single pass compiler due to future references (or) Jumps.

→ A pass compiler should be designed properly inorder to save compilation time.

→ Pascal and C uses a Single pass compiler.

(*) Advantages of Pass:

→ A two-pass compiler, where one pass takes the input and generates an intermediate code and passes it to the two pass which then generates the target code, has an advantage over the time spent in generating intermediate code.

→ In a Single-pass compiler, the program is read only once.

→ The execution time for a two-pass compiler is very short.

(*) Disadvantages of pass:

→ For a single-pass compiler, the memory required is more.

→ Code generation by a single pass compiler is not efficient.

→ Complexity ~~and~~ in designing a pass-based compiler is more.

Scanned by CamScanner

phase of a compiler:

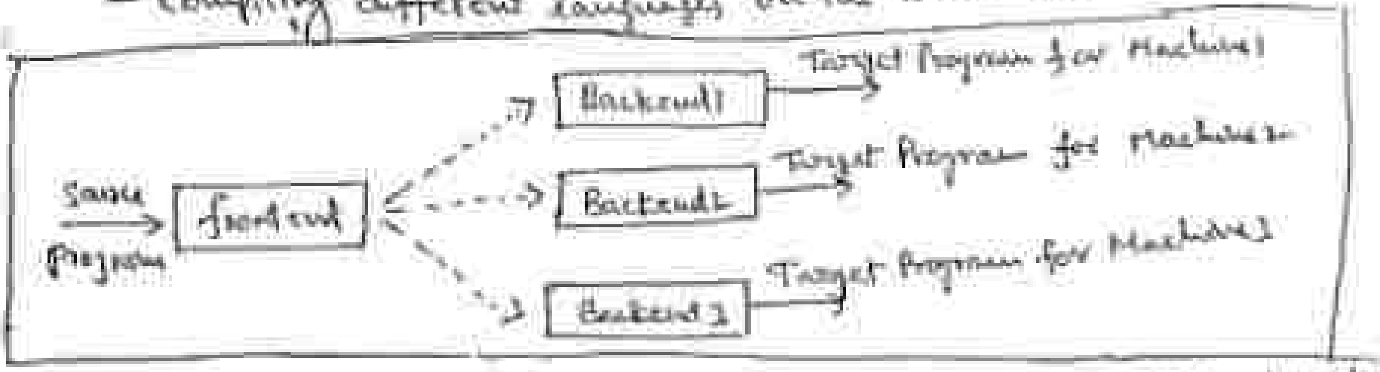
-> A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.

-> There are two parts

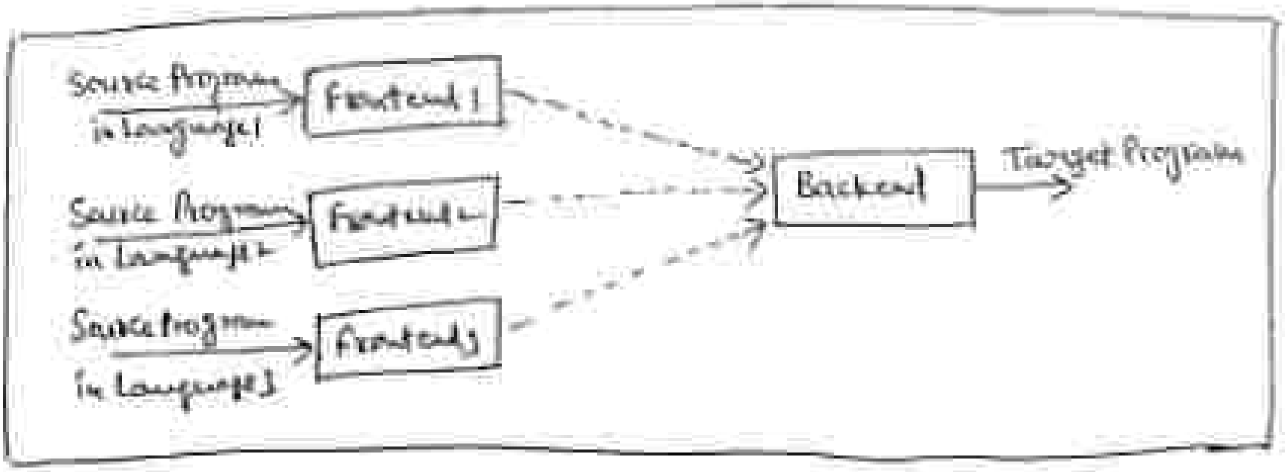
- Analysis phase (front-end)
- Synthesis phase (Back-end)

-> By dividing the ~~Compiling~~ compiler into two parts, we can do two things

- Compiling same source language on different machines.
- Compiling different languages on the same machine.


Source Program
frontend
Backend1
Backend2
Backend3
Target Program for Machine1
Target Program for Machine2
Target Program for Machine3

Compiling same source language on different Machine


Source Program in language1
Frontend1
Source Program in language2
Frontend2
Source Program in language3
Frontend3
Backend
Target Program

Compiling different languages on the same machine

-> These two things are possible because front-end is language dependent and Machine Independent and Backend is language Independent and machine dependent.

The intermediate code generator converts the tree from semantic analyzer to an intermediate representation of the source program. For this, it uses temporary variables. For the given statement, the intermediate code generator uses three temporary variables t_1, t_2 and t_3.

An optional phase in the compiler is the code optimizer. Its job is to optimize the code. Optimization refers to minimize the number of instructions, operations, variables or temporaries etc., without changing the meaning of the statements generated by the intermediate code generator. In figure, the number of temporary variables is reduced from three to one. Moreover, the "inttoreal" function is avoided and the value 60 is directly used as 60.0. The number of instructions is also reduced from four to two.

The final phase in the compiler is the code generator. It generates assembly code or relocatable machine code. Since, the given instruction is of floating-point type, the code generator uses float version of the instructions such as MOV, MUL and ADD. A constant is represented with a preceding "#". The most important task done by the code generator is assigning variables to registers.

Q15. Discuss in detail about compiler construction tools.

Answer :

A compiler make use of some specialized tools inorder to implement different phases of compilation. These specialized tools are also known as compiler-compiler or compiler-generators or translator writing system. These tools provides automation to compiler design.

The following are some of the compiler-construction tools.

1. Scanner generators
2. Parser generators
3. Syntax-directed translation engines
4. Code-generators
5. Data-flow analysis engines and
6. Compiler-construction toolkits.

1. Scanner Generators

It takes input in the form of regular expression and generates output as a lexical analyzer. For example, LEX scanner generator used in unix.

2. Parser Generators

It takes input in the form of context free grammar and generates output as a syntax analyzer. For example, YACC parser generator used in unix.

3. Syntax-Directed Translation Engines

It takes input in the form of parse tree and generates output as an intermediate code by translating each node present in the parse tree.

4. Code-Generators

It takes input in the form of intermediate code and translates into its equivalent machine code by making use of template matching technique.

5. Data-Flow Analysis Engines

It helps in collecting information regarding the flow of values from one part of a program to another part of a program. This is considered as an crucial step in performing good code optimization.

6. Compiler-Construction Toolkits

It offers an integrated set of routines in building different compiler's phases.

3

The Evolution of Programming Languages

Q16. Explain the classification of programming languages.

Answer :

Based on the model of computation all the existing languages have been classified into families as listed below.

1. Declarative languages
2. Imperative languages.

1. Declarative Languages

The languages that focus on "what the computer is to do" comes under declarative language. It is some times called "higher level" language. This type of languages are more beneficial to the programmers and less beneficial to the implementors. Declarative languages have been categorized into the following subclasses.

(i) Functional languages
(ii) Dataflow languages
(iii) Logic, constraint-based languages.

(i) Functional Languages

A functional or applicative language is one in which the primary means of making computation is by applying functions to given parameters. Programming in functional languages can be done without using any kind of variables, assignment statements and iterations. Thus, the functional languages, instead of focusing on the changes in state, emphasizes on the function that must be applied to the initial machine state. In these languages, the programmers need not be concerned with the variables because the memory cells are not abstracted into the language. The development of a program proceeds by generating the functions from previously created functions to obtain more complex functions that can manipulate the initial data to obtain the final result.

Syntax

The syntax of these languages is shown below.

function_n(..... function_2(function_1(data)) ...)

Je

(List processor) & ML(Meta language)

Advantages

- Functional languages have simple syntactic and semantic structures.
- Concurrent execution of programs is easy, as the programs are first converted into graphs which in turn can be executed through graph-reduction methods.
- Less labor is required for constructing the programs.
- Cooperation synchronization is not the burden of a programmer.

Disadvantage

Less efficient than the imperative languages.

Example

Lisp(List processor), Haskell ML(Meta Language).

i Dataflow Languages

These languages employ a computational model based on the flow of information (i.e., tokens) between the functional nodes. The model is inherently parallel in which as soon as input tokens arrived nodes get trigger this is a concurrent process.

Example

This category of language include Id, Val.

ii) Logic. Constraint-based Languages

In rule-based (logic) languages, no specific order is followed in specifying rules. Thus, the language implementation system must select some order of execution that leads to the appropriate result. These languages can execute an action based on the satisfaction of certain enabling conditions. PROLOG is the most common rule-based (logic) programming language which consists of a class of predicate logic expressions as its enabling conditions. Execution of a rule based language is similar to an imperative language except that the statements are not sequential. Enabling conditions determine the order of execution.

Syntax

The syntax of such languages is similar to,

```
enabling conditioq_1 → Action_1
enabling condition_2 → Action_2
enabling condition_3 → Action_3
  :
  :
enabling condition_n → Action_n
```

2. Imperative Languages

The languages that *focus* on "how the computer should do it" comes under imperative languages.

Imperative languages have been categorized into the following subclasses.

(i) Von Neumann Languages

These languages are the most successful and familiar whose computation model is on the basis of variable modification. These are statement particularly assignments based languages.

Example

Languages in this category include Ada, Fortran 83, C.

(ii) Scripting Language

These are the subset of VonNeumann languages but differ by the components "giving together" which were independent program in the beginning.

Many of the scripting languages were initially developed for some specified purposes. For example, csh and bash was developed for job control (shell) programs as the input languages. AWK was developed for the manipulation of text. JavaScript and PHP were designed for the purpose of creating web pages. Some other languages such as Ruby, perl, python, and Tcl were intentionally developed for general purposes.

(iii) Object-oriented Languages

In object oriented programming, everything is considered as objects. In this case complex data objects are created and set of functions that can operate on those created objects are designed. Complex objects can inherit properties of the simpler objects. By creating concrete data objects, the object oriented programs can gain efficiency over imperative languages.

Example

C++, Java, Ada and Small talk.

Advantages

- Complexity can be easily managed by using objects.
- Security of data is more when compared to procedural and functional oriented languages.

4 The Science of Building a Compiler

Q17. Define the term code optimization. Discuss briefly the design objectives of compiler optimization.

Answer :

Code Optimization

The term 'code optimization' refers to program transformation method in the synthesis phase that tries to generate a machine code which can runs faster using fewer resources like CPU and memory.

Objectives of Compiler Optimization

The design objectives or goals of compiler optimization process are as follows,

1. **Correctness**

   The compiled program in any way need to maintain the meaning of the program i.e., it requires correct optimization.

2. **Performance**

   The optimization need to improve or enhance the performance of the program. In this, performance means execution speed of the program.

3. **Compilation Time**

   The time required for compiling the program need to be acceptable so as to support development and debugging cycle.

4. **Maintenance Cost**

   In general compiler systems involve complexity. Hence, the systems need to be simple so that the engineering effort and maintenance costs of the compiler are manageable to solve critical problems.

Q18. Give the classification of code-optimization.

Answer :

The classification of code optimization is based on,

(i) Level of code

(ii) Programming language

(iii) Scope.

(i) **Level of Code**

(a) **Design Level**

Efficient resources and an appropriate algorithm can enhance the efficiency of code.

(b) **Source Code Level**

The object code performance can be improved by the user by changing the program and modifying the algorithm.

(c) **Compile Level**

The program can be improved by the compiler by improving the loops and performing optimization on procedure calls and address calculations.

(d) **Assembly Level**

The code optimization can be performed by the compiler depending on machine architecture. Which is based on available registers as well as appropriate addressing modes.

(ii) **Programming Language**

(a) **Machine Independent**

1. It is dependent on the instruction set and addressing modes to be used.
2. The efficiency of the program is improved by allocating sufficient number of resources.
3. Intermixed instructions with data increases the speed of execution.
4. Intermediate instructions are used wherever necessary.

(b) **Machine Dependent**

1. It is independent of the target machine but depends on the source language characteristics.
2. The efficiency of the target code is improved by using appropriate program structure.
3. Elimination of dead code increases the speed of execution.
4. Identical computations are moved to one place, thus avoiding the repeated computation of an expression.

(iii) **Scope**

(a) **Local Optimizations**

These are the optimizations carried out within a single basic block. This technique do not require the information regarding the data and flow of control. Thus, implementation of this technique is simple.

(b) **Global Optimizations**

These are the optimizations carried out across basic blocks, instead of single basic block. This analysis is also known as data-flow analysis. In this technique, additional analysis is required across basic blocks. Thus, implementation of this technique is complex.

Application of Compiler Technology

Q19. List all the applications of compiler technology and explain each of them in detail.

Answer : Model Paper-III, Q7(a)

Applications of Compiler Technology

Compiler technology is used for,

1. Implementing high level programming languages
2. Optimizing the computer architectures
3. Designing new computer architecture
4. Translating program.

1. **Implementing High Level Programming Languages**

Compilers are used for converting high level programming languages into machine level languages. The major difference between both the languages is that the former is easier to write and is less efficient where as the latter is harder to write but is efficient. Moreover, low level languages have high possibility of errors and requires a lot of maintenance. However, the inefficiency factor measured while using high level languages can be by passed by optimizing the compiler which employs certain techniques.

ntly, there are many programming languages are capable of providing high level of abstraction. these languages include C, C++, Java. Both C and nguages have their own language specific features ave made them predominant over one another . All a languages are object oriented where in the key concepts clude data abstraction and inheritance. Optimizing compiler n such languages means that , the optimized compiler should be capable of performing its operation efficiently across the procedural boundaries of the source programs.

Java is another object-oriented based programming language that has the following features.

(i) It is type-safe.

(ii) It checks whether the arrays are defined within the array bounds.

(iii) It doesn't supports the concept of point.

(iv) It doesn't perform pointer arithmetic.

Despite of all these features, Java incurs a run-time overhead which is reduced by performing compiler optimization.

2. Optimizing the Computer Architectures

Compiler technology is used in optimizing computer architectures using the following techniques.

(i) Parallelism

(ii) Memory hierarchies.

(i) Parallelism

In the present era, many microprocessors depends on instruction level parallelism, which is abstracted from the programmer. In this sort of parallelism multiple instructions are executed irrespective of their order, so as to enhance the efficiency. However, it is the responsibility of a hardware-scheduler to decide the order of instruction to be executed.

Instruction level parallelism is explicitly defined in the instruction set one such machine is VLIW(Very Long Instruction Word) that executes multiple operations of an instruction parallely. Intel IA64 fall under this category of machine. Many compiler techniques are developed with an intention of generating code automatically for these machines, such compilers,

(a) Hides the details of parallelism.

(b) Distributes the computation among multiple machine.

(c) Minimize synchronization among the processors.

(ii) Memory Hierarchies

Computer programmers sometimes prefer to use smaller but fast memories and sometimes larger but slow memories. The requirement for a specific memory depends on the type of computation. These memory requirements led to the development of memory hierarchy. Memory hierarchies are found in each and every machine that is being developed. The major notion of this hierarchy is that the storage devices that is faster in speed but smaller in size is kept nearer to the processor and the device that is slower in speed but larger in size is kept away from the processor . The advantage of existence of memory hierarchy is that it enhances the performance of the machines.

Memory hierarchy consists of the following devices.

(a) Less number of registers that stores hundred of bytes.

(b) Different level of caches that stores kilobytes to mega bytes.

(c) Main memory that stores megabytes to gigabytes.

(d) Secondary storage that stores gigabytes and more

Registers are explicitly managed by the software. where as the caches and main memory are managed by hardware(which are not effective in certain scenarios) It is even possible to enhance the level of effectiveness of memory hierarchy by,

(a) Modifying the layout of the data.

(b) Changing the order of instructions that are accessing the data.

(c) Modifying the layout of the code.

3. Designing New Computer Architecture

Unlike, the earlier designs of computer architectures, the modern computer architecture have compiler embedded in the machines. This is because, the performance of a system is depended on the way the compiler performs its operation, rather than on its raw speed. The following are few modern computer architectures.

(i) RISC

(ii) Specialized architectures.

(i) RISC

Reduced instruction set computer is a kind of micro-processor architecture, that makes use of small, highly-optimized set of instructions. It is designed to recognize a relatively small number of computer instructions, so that it can perform its operation at higher speed.

Advantages

❖ It have a clock per instruction of one cycle.

❖ It supports the pipelining concept that allows execution of parts simultaneously.

❖ It contains large number of registers to prevent interactions with memory.

(II) **Specialized Architectures**

The different architectures designed over the past decades include,

(a) Data flow machines.

(b) Vector machines.

(c) VLIW(Very Long Instruction Word) machines.

(d) SIMD(Single Instruction Multiple Data) arrays of processors.

(e) Systolic arrays.

(f) Multiprocessors with shared/distributed memory.

4. **Translating Program Code**

Compiler technology is used for translating high level programming code into machine level programming code. This translation helps in,

(i) Performing binary translation

(ii) Synthesizing the hardware

(iii) Interfacing database query.

(i) **Performing Binary Translation**

The program translation technique is used for translating the program written in binary form for one machine into a form that can be used by another machine. Thereby, allowing a machine to execute the program irrespective of the instruction set. The binary translation technology is used so as to,

(a) Increase the availability of software for their machines.

(b) Provide backward compatibility.

(ii) **Synthesizing the Hardware**

Hardware-synthesis tools perform automatic translation of RTL description into gates. These gates are then mapped into transistors followed by the physical layout.

(iii) **Interfacing Database Query**

Database queries consists of predicates that are interpreted into commands. This interpretation is done for searching a record in a database that satisfies the predicate.

Programming Language Basics

Q20. Discuss about programming language basics.

Answer : Model Paper-IV, Q2(a)

Programming Language Basics

The basic concepts involved in the study of programming language are,

1. Difference between static/Dynamic policy
2. Meaning of environment and state
3. Significance of static scope and block strutures
4. Meaning of dynamic scope
5. Methods of passing parameters.

1. **Difference Between Static/Dynamic Policy**

While designing programming languages, it is necessary to decide whether to opt static policy or dynamic policy. Static policy allows the compiler to decide regarding an issue at compile time where as the policy that allows the same decision at runtime is referred as dynamic policy. Scope is one of the major issue to be concentrated while designing a programming language.

A scope can be either static or dynamic. Static scope is used when it is possible to determine the scope of declaration just by analyzing the program. On the other hand, if it is not possible to determine the scope of declaration than in such situation dynamic scope is used by the languages.

2. **Meaning of Environment and State**

Before, designing the programming language it is necessary to ensure that the language being designed has the ability of mapping the data names with their respective data values. This mapping is done using the following two stage mapping.

(i) Environment

(ii) State.

(i) **Environment**

It is a mapping where in the data names are mapped to their respective location in the memory.

(ii) **State**

It is a mapping where in the location are mapped to their respective data values. This two-stage mapping is dynamic in nature.

3. **Significance of Static Scope and Block Structures**

Static scoping determines the declaration of a name only by examining the program text. The languages like Pascal, C and Ada use static scoping. The scope of the variable is determined with the help of blocks with in the program.

Example

```
main( )
{
      int a = 1, b = 2, d;
      {
            int a = 2, c;
            {
                  c = a + b;
            } //block A
            d = a + b;
      }// Block B
} //Block C
```

In the above example, the scope of variable 'a' is declared in block 'C' as a = 1 is of whole block 'C'. But the block 'B' is out of scope of this declaration it redeclares the variable as a = 2. Hence, the value of c will be 4 and that of 'd'

...ecture

A block is a part of a program that contains declarations & enclosed in { }. For example,

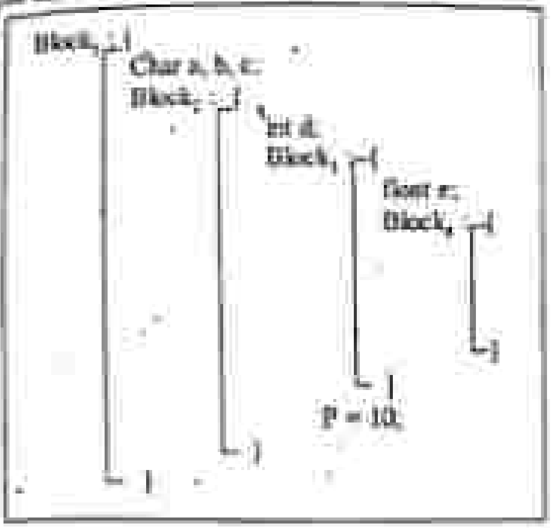


Figure (1): Block in a Program

A block structured language is the one that allows a block or a number of blocks inside one block where the scope of declaration is limited to block in which declaration is made.

Block information can be,

- ❖ Block number given by Block #
- ❖ Enclosing or outer block given by outer Block#.
- ❖ Number of declarations in a block.
- ❖ First declaration in a block.

In the above figure,

The outer block# for the 1st Block is 1 indicating Block₁ is the first block.

Block₁ has three declaration with a being the first one from left to right in the same declaration line.

Thus, the entries for symbols in the symbol table should be in the order of declaration in blocks.

4. Meaning of Dynamic Scope

Dynamic scoping determines the declaration of a name time with the help of current activations. The languages like snobol use dynamic scoping. The scope is determined with the help of current activations.

Example

```
#define a(d + 1)
void x( )
{
        printf("%d",a);
}
void y( )
{
        int d = 0;
        printf("%d",a);
}
void main( )
{
        x( );
        y( );
}
```

In the above example, the value of the variable 'a' is determined at runtime during the execution of the functions x( ) and y( ). When the function x is called, value of d is same as declared globally thus, the printf statement prints the value of macro 'a' as 3. But upon execution of function y( ) it prints the value of macro 'a' as 1. Since, at this time it refers to the declaration of d = 0 in function y( ).

5. Methods of Passing Parameters

Parameter passing method are the method which provides the means of communication between a calling and a called procedure. The methods that associate actual parameters with the formal parameters are as follows,

(i) Call-by-value

(ii) Call-by-reference

(iii) Copy-restore

(iv) Call-by-name.

(i) Call-by-value

Call-by-value is the simplest method of parameter passing, in which caller evaluates the actual parameters and pass their values to the called procedure as formal parameters. In the called procedure, formal parameters are treated as local names whose storage is in the activation record of the called procedure. Since formal parameters behaves as constant values during the execution of a procedure, performing any operation on formals do not affect the values in the activation record of the caller. "C" uses call-by-value for passing parameters and it is the default method in Pascal and Ada.

For example, consider the following Pascal program that swaps the values of two variables.

```
Program example(in, out);
var i, j : integer;
procedure swap(var a, b : integer);
    var t : integer;
    begin
        t : =a;
        a : = b;
        b : = t
    end;
    begin
        i : = 100;
        j : = 50;
        swap(i, j);
        writeln('i value =' i);
        writeln('j value =' j);
  end
```

In this example, the program calls the procedure swap (i,j) by call-by-value. The values of i and j are copied into the formal parameters a, b. The procedure call changes the values of the local variables a, b and t of swap but these changes will be lost as soon as the control returns to the caller at the end the activation record of swap is deallocated. Hence, the call-by-value method does not effect the values in the activation record of the caller.

The call-by-value method can affect its caller if it accesses non local names and pointers are passed as values.

(ii) **Call-by-reference**

Call-by-reference is a method in which caller passes the storage location of the variable instead of passing the value of a variable to the called procedure. So the formal parameter becomes an alias for the actual parameter. That is, the caller copies the addresses of actual parameters into the activation record of the called procedure. Therefore any changes made to the formal parameters affect the values of the actual parameters. Many programming languages use call-by-reference method. In FORTRAN77 parameter passing is call-by-reference. Pascal achieves pass-by-reference with the use of keyword var and C++ by using the symbol & in the parameter declaration. Arrays are always passed by reference. Call-by-reference is also called *call-by-address* or *call-by-location*.

For example, a call to swap procedure as swap (i, a[i]) results in following steps:

(a) The address of actual arguments i and a[i] are copied into the locations say l_1 and l_2 in the activation record of the called procedure.

(b) Assign the value of a temporary variable, say t, to the value of the location pointed to by l_1, that is set t = i.

(c) Assign the value of address pointed by l_2 to value of address pointed by l_1, that is, set i = a[i]. Where i_0 is the initial value of i.

(d) Assign the value of the location pointed to l_2 equal to the value of t, that is set a [i_0] = t.

The difference between call-by-value and call-by-reference is that unlike call-by-value, call-by-reference does not make copies of the actual parameters. Therefore, it is important to pass an argument by call-by-reference when the value to be passed is a large structure but prohibit any changes to the values of the actual parameters.

(iii) **Call-by-Value-Result (Copy-in-copy out or Copy_restore)**

copy-in copy-out or call by value_Result is a method. It is between call-by-value and call-by-reference. Like call-by-value, the calling procedure evaluates the values of actual arguments and passes them to the called procedure. This corresponds to *copy-in* which copies the values of actual arguments into the storage for the formal parameters that is in the activation record of the called procedure. After the control returns to the calling procedure, the current values of the formal parameters are copied back into the locations of the actual arguments. This corresponds to *copy-out* which copies the final values of the formal parameters into the storage for the actual arguments that is in the activation record of the calling procedure. Some FORTRAN implementations uses copy-restore method.

The following program shows the differe[illegible] call-by-reference and copy-restore. This is possible if the procedure can access the location in the activation [illegible] caller in more than one way.

```
Program [illegible](in, out);
    var x : integer;
    procedure fun(var i : integer);
    begin
        i : = 20;
        x : = 30;
    end;
begin
    x : = 10;
    fun(x);
    writeln(x);
end
```

In the program, the procedure fun( ) can access the value of x as a non local and through the formal parameter i. Under value-result the call fun(x) copies.

The value of x to i i.e., i = 10 and then change.

The value of i to 20, when the call returns.

The final value of i is copied into x.

Therefore, the final value of x is 20 instead of 30. Under call-by-reference, the value of x is 30 because the changes made to i and x immediately affect x.

(iv) **Call-by-name**

In call-by-name, whenever a procedure is called the body of called procedure is substituted in the body of the calling procedure with substitution of the actual parameters for the formal parameters. The method is called *macro-expansion* or *in-line expansion*. To prevent any name clashes, the locals of called procedure are kept separate from the locals of calling procedures and renaming the locals of called procedure before the macro-expansion is done.

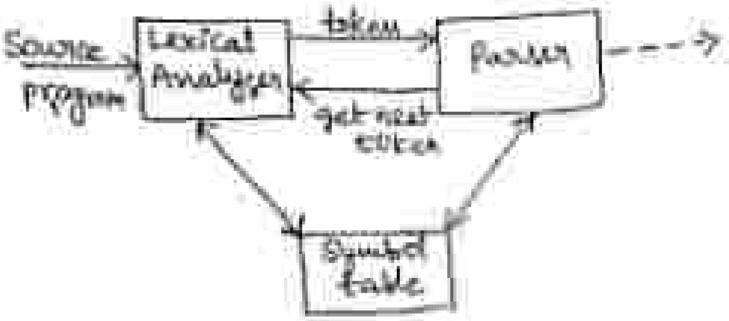
Call-by-name is not a popular method of parameter passing. But it is suggested if the running time of a program is to be reduced. When a procedure call occurs, setting up an activation record of a procedure incurs certain cost. First the space is to be allocated for it, the machine status must be saved, setup the links and then transfer the control to the called procedure. This is an overhead when procedure body is small. In this case, it is efficient to use the in-line expansion of the procedure body in the code of the calling procedure.

EXICAL ANALYSIS:- (The Role of lexical Analysis)

→ The lexical analysis is the first phase of a compiler.

→ Its task is to read the input characters and produce a sequence of tokens as output.

→ The Interaction of Lexical analyzer with parser is



→ Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

(*) Need for lexical Analysis:-

→ The purpose of splitting the analysis of a source program into three phases, i.e, lexical analysis, syntax analysis and semantic analysis is to simplify the overall design of the compiler.

→ In lexical analysis, it is easier to specify the structure of tokens than the syntactic structure of the source program.

→ Lexical Analyzer keeps track of line numbers, producing an output listing if necessary, stripping out white space such as redundant blanks and tabs and deleting comments.

(*) Tokens, Patterns and lexemes:-

→ Token is a sequence of characters having a collective meaning.

→ Pattern is a rule describing the set of strings in the input for which the same token is produced as output (or)
It is a rule describing the set of lexemes that can represent a particular token in the source program.

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

→ In most programming languages keywords, operators, identifiers, constants, literal strings and punctuation symbols such as pattesis, commas and semicolons are treated as tokens.

Ex:- if (a < b)

lexeme	Token	Pattern
if	Identifier	if
(	left Paranthesis	(or)
a	identifier	letter followed by letter or digit
<	relational-op	< or <= or > or >= or <> or ==
b	Identifier	letter followed by letter or digit
)	Right Paranthesis	(or)

(*) Attributes for tokens :-

-> Lexical analyzer provides additional information to distinguish between the similar type of pattern that match a lexeme.

-> The lexical Analyzer collects the attributes (properties or features) of tokens as the additional information.

-> A token has a single attribute, i.e, a pointer to the symbol table entry in which the information about the token is kept.

Ex:- E:=M*C**2

lexeme	Tokens	Attribute values
E	id	Pointer to symbol-table entry for E
:=	assign-op	-
M	id	Pointer to symbol-table entry for M
*	Mul-op	-
C	id	Pointer to symbol-table entry for C
**	exp-op	-
2	number	2

(3) Input Buffering:-

-> There are 'three general approaches to the implementation of a lexical analyzer.

i) Use a lexical analyzer generator such as lex compiler, to produce the lexical analyzer from a regular-expression-based specification.
In this case, the generator provides routines for reading and buffering the input.

i) Write the lexical analyzer in a conventional system-programming language using the I/O facilities of that language to read the input.

(iii) Write the lexical analyzer in assembly language and explicitely manage the reading of input.

→ The lexical analyzer is the only phase of the compiler that reads the source program character-by-character.

Buffer pairs:-

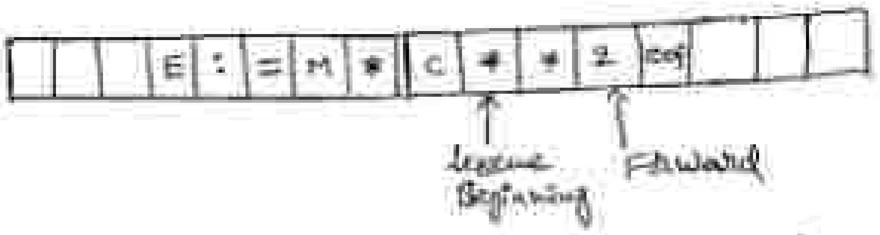
→ The lexical analyzer scans the characters of the source program one at a time to discover tokens.

→ There are many schemes that can be used to buffer input.

→ One class of schemes here,

- A Buffer divided into two N-character halves.
- 'N' is the number of characters on one disk block.

Ex:- 1225



An Input Buffer in Two Halves

- We read 'N' input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character.
- If fewer than 'N' characters remain in the input, then a special character eof is read into the buffer after the input characters.
- eof marks the end of the source file and is different from any input character.

→ Two pointers to the input buffers are maintained.

→ The string of characters between the two pointers is the current lexeme.

(i) lexeme beginning ~~token~~ pointer

(ii) forward (or) lookahead pointer.

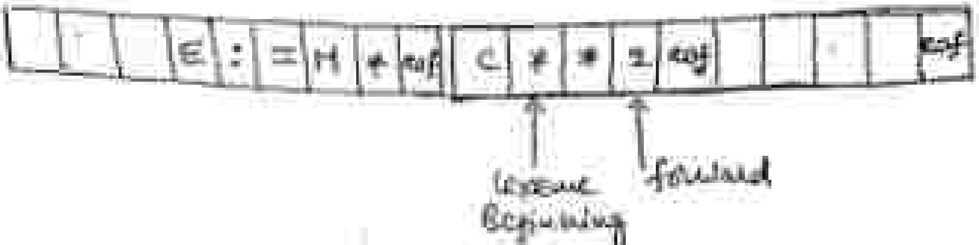
Sentinels :-

→ Except at the ends of the buffer halves, the code requires 2 tests for each advance of the forward pointer.

→ We can reduce the two tests to one if we extend each buffer half to hold a sentinel character at the end.

→ The Sentinel is a special character [eof] that cannot be part of the source program.

→ The Same buffer management with sentinals added is



Sentinels at end of each Buffer Half

(4) Token Specifications :-

(i) Alphabets, Strings, Languages and Language operations.

(ii) Regular Expressions.

(iii) Regular Definitions (or) Regular Grammars.

(iv) Notational Shorthands (or) Extensions of Regular Expressions.

(ii) Regular Expressions

→ A Regular Expression is any well-formed formula constructed over union, concatenation and closure.

→ Each regular expression(r) denotes a language L(r).

→ It is having a set of defining rules.

~~[illegible]~~

- 'ε' is a regular expression that denotes {ε}, i.e, the set containing the empty string.
- If 'a' is a Symbol in 'Σ', then 'a' is a regular expression that denotes {a}, i.e, the set containing string 'a'.
- Suppose 'r' and 's' are two regular expressions denoting the languages L(r) and L(s). Then,

→ r+s is a regular expression denoting L(r) ∪ L(s)

Scanned by CamScanner

→ rs is a regular expression denoting $L(r)\cdot L(s)$

→ r^* is a regular expression denoting $L(r^*)$

A language denoted by a regular expression is said to be <u>regular set</u>.

→ There are three operations on regular expressions.

a) Union, denoted by + (or) |

b) Concatenation, denoted by dot '(·)' (or) no symbol.

c) closure, denoted by *.

<u>Precedence of operators</u> :-

→ Unary operator '*' has the highest precedence and is left associative.

→ concatenation operator has the second highest precedence and is left associative.

→ Union operator '+' has the lowest precedence and is left associative.

<u>Ex</u>:- ab^*+a has the steps

b^*

ab^*

ab^*+a

<u>Examples of Regular expressions</u>:-

→ Let $\Sigma = \{a, b\}$

*) The regular expression $a|b$ denotes the set $\{a, b\}$

*) The regular expression $(a+b)(a+b)$ denotes $\{aa, ab, ba, bb\}$, the set of all strings of a's and b's of length two.

*) The regular expression a^* denotes $\{\varepsilon, a, aa, aaa, \ldots\}$, the set of all strings of zero (or) more a's.

*) $(a+b)^*$ denotes the set of all strings containing zero (or) more instances of a's (or) b's.

Algebraic Laws for Regular Expressions :-

Axiom	Description
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$(rs)t = r(st)$	Concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity element for concatenation
$r^* = (r\|\epsilon)^*$	relation between * and ϵ
$r^{**} = r^*$	* is idempotent

(iv) Regular Definitions :-

→ If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\boxed{\begin{array}{l} d_1 \rightarrow r_1 \\ d_2 \rightarrow r_2 \\ \text{-- -- -- --} \\ \text{-- -- -- --} \\ d_n \rightarrow r_n \end{array}}$$

where, each d_i - distinct name

r_i - Regular expressions over the symbols and the previously defined names.

Ex:- Regular definition for identifier in pascal language.

letter → $A|B|\cdots|Z|a|b|\cdots|z$.

digit → $0|1|\cdots|9$

id → letter (letter | digit)*.

(v) Notational Shorthands :-

a) One (or) more Instances

b) Zero (or) One Instance

c) character classes.

a) One (or) more Instances :-

→ The operator '+' means "one (or) more Instances of"

→ If 'r' is a regular expression that denotes the language $L(r)$, then $(r)^+$ is a regular expression that denotes the Language $(L(r))^+$

Ex:- $a^+ = \{a, aa, aaa, \ldots\}$

b) Zero (or) One Instance :-

→ The operator '?' means "Zero (or) One Instance of".

→ The notation r? is a shorthand for $r|\epsilon$.

Ex:- $a? = a|\epsilon$.

c) character classes :-

→ It uses the symbols [ ]

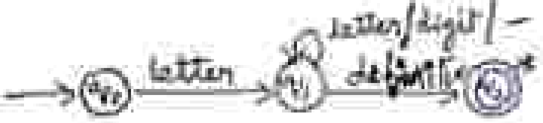
→ The notation [abc] where a,b,c are alphabet symbols denotes the regular expression a|b|c.

→ An abbreviated character class such as [a-z] denotes the regular expression a|b| .. |z.

→ A pascal ~~identifier~~ identifier is represented by a regular expression as

$$[A\text{-}Z\,a\text{-}z][A\text{-}Z\,a\text{-}z\,0\text{-}9]^*.$$

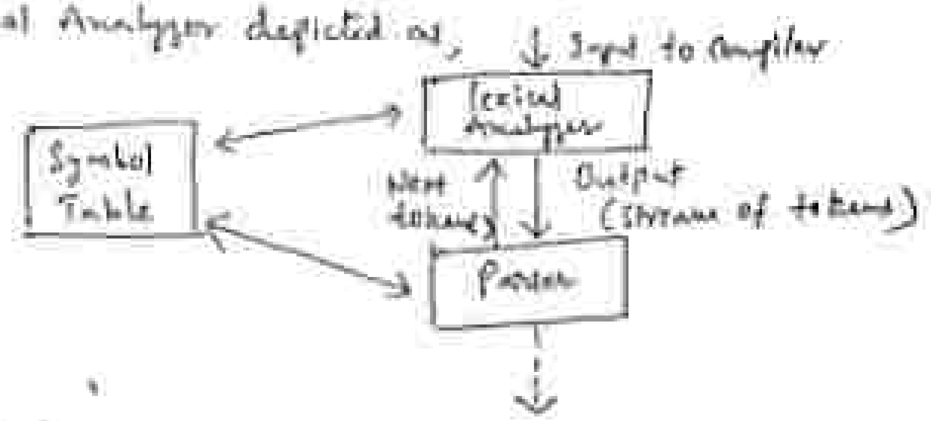
→ The transition diagram for an identifier is



(3) Lexical Analysis Vs Parsing:

→ The lexical Analyzer being the first phase of a compiler reads(or) scans the input to the compiler and divide the input into a number of small parts of a string (token)

→ Lexical Analyzer depicted as,



→ It is invoked by the parser.

→ There are number of reasons for which the lexical Analyzer is seperated from Syntax Analyzer.

(i) Division of lexical Analyzer from Syntax Analyzer improves the efficiency of the compiler.

(ii) By the division of lexical analyzer and Syntax analyzer into two phases, the designing of a compiler is made very easy

(iii) Token generation and then storing the input token with its attributes in symbol table is not an easy task.

(iv) The division reduces compilation time

(v) The division improves portability of the compiler.

(4) Lexical Errors:

→ A lexical Analyzer reads input string and generates tokens.

→ It may encounter the following types of errors.

(i) A misspelled identifier

(ii) An identifier with over than specified or defined length

→ These errors can be caused due to the following reasons.

- An extra character than specified or defined length
- An invalid character
- A missing character
- Misplaced characters.

Example :

```
main()
 {
  printf("Hello)
 }
```

In this code, 2 syntax errors.

(i) double quote missing right side of string "Hello".

(ii) Semicolon missing in printf() statement.

(10) Reserved words and Identifiers: (Recognition of Tokens)

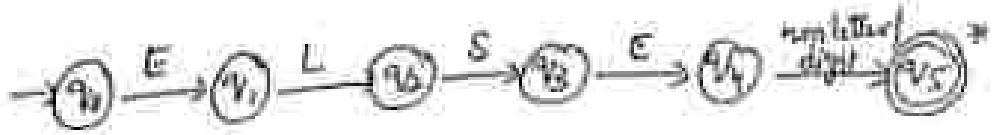
→ A Reserved word is also called as keyword.

→ Keywords are the words whose meaning has been already explained in the compiler.

→ Example keywords are,

auto, break, case, char, const, continue, do, default, else, enum, if, for, while, ...

→ The transition diagram for ELSE is



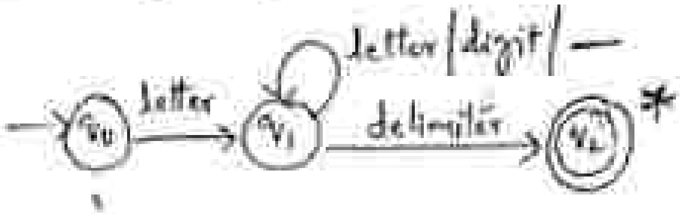
→ An identifier is any user-defined name

example: a, b, c, sum, sum(), print(), display(), ...

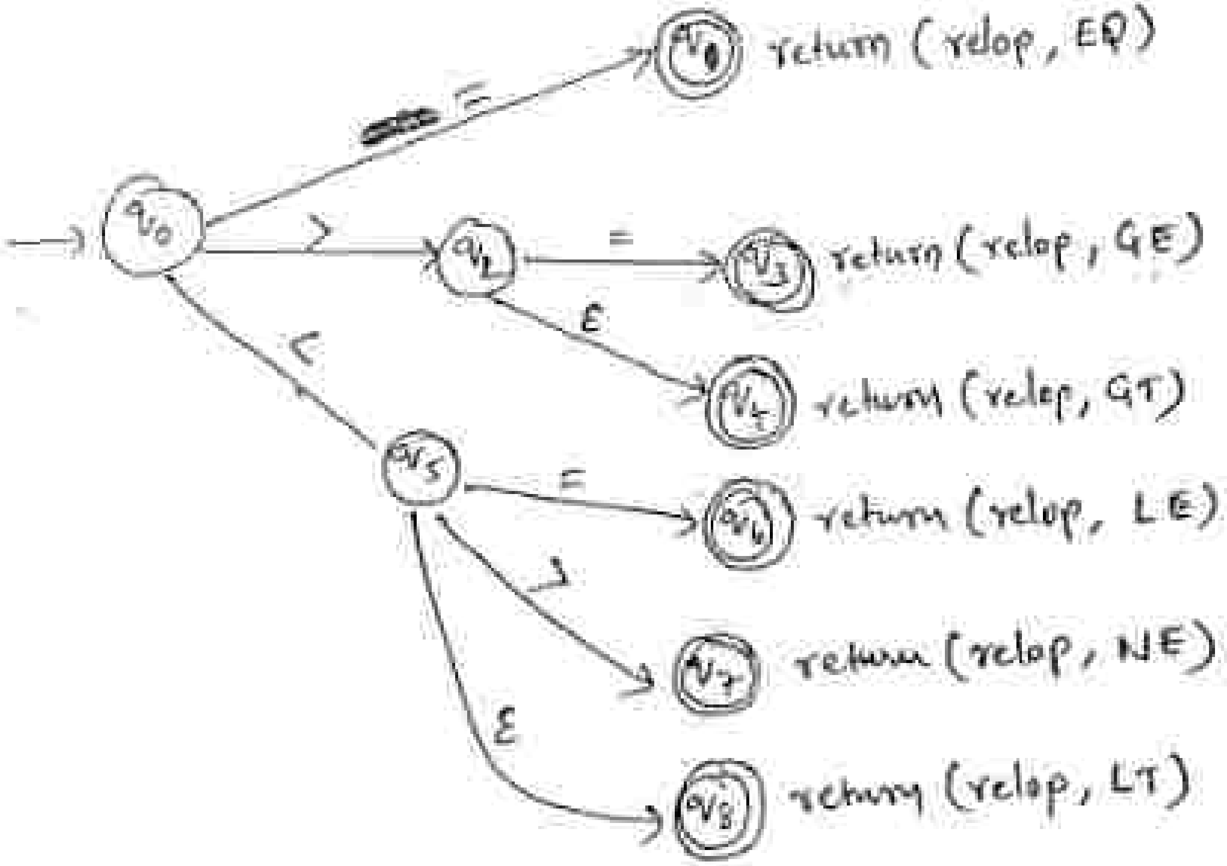
→ An identifier cannot start with a digit.

- Keywords (or) Reserved words can not be represented as identifiers

→ Transition diagram for identifier is



Transition Diagram for Relational operators:



LEX Tools :-

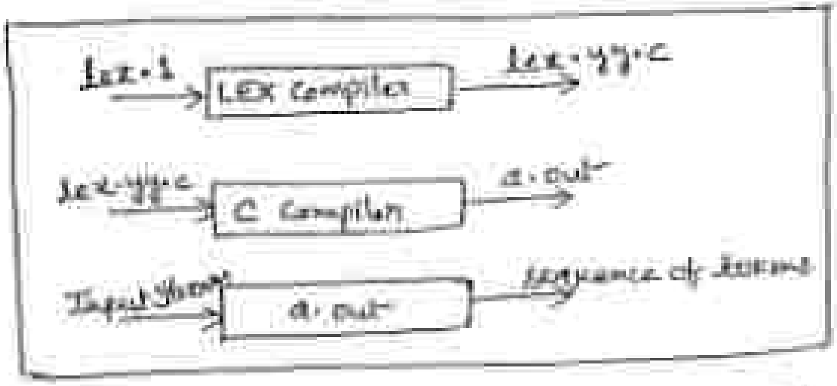
→ There is a wide range of tools for construction of lexical Analyzer.

→ The majority of these tools are based on regular expressions

→ One of the traditional tools of regular expressions is LEX or lex

→ We refer to the tool as the LEX compiler and its input specification as the lex language.

→ Creating lexical Analyzer with lex is represented as,



→ A specification of the lexical analyzer is prepared by creating a program lex.l in the lex language.

→ The lex.l is run through the lex compiler to produce a 'c' program lex.yy.c

→ The program lex.yy.c consists of 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.

(*) LEX Specifications :-

→ A lex program consists of three parts

```
declarations
 %%
translation rules
 %%
auxiliary procedures
```

→ The declaration section includes, the declaration of

- variables
- manifest constants and
- regular definitions

→ A manifest constant is an identifier that is declared to represent a constant

→ The regular definitions are statements and are used as components of the regular expressions appearing in the translation rules.

→ The Translation rules of a LEX Program are statements of the form

```
P1 {action 1}
P2 {action 2}
 - - - - - -
 - - - - - -
 Pn {action n}
```

- where P_i is a regular expression.

→ Auxiliary procedures are needed by the actions and can be seperately compiled and loaded with the lexical Analyzer.

(*) EXAMPLE

(i) Write a lex program to identify comments in the program.

```
%{ /* Lex Program to identify comments in a program */ %}
Comments %%         {/* (letter| digit | whitespace)+ */}

{comments}          {/* No action is taken, No value is returned*/}
 %%
install_id()        {/* Install lexemes in the Symbol table */}
install_num()       {/* Install numbers as lexemes in the
                         symbol table */}
```

LEX PROGRAM to Recognize the decimal number :

-> LEX Program is a tool that is used to generate lexical analyzer (or) Scanner.

-> The tool is often refered as lex compiler and its input Specification as LEX language.

-> LEX Program to recognize decimal numbers is,

```
    %{
       /* The program to recognize decimal number */
       #include <stdio.h>
       int i;
     %}
  %% [0-9]+[.]
     {
      for(i=0; i<=yylength; i++)
           {
            if(yytext[i]='.')
               {
                printf("%s is a decimal number", yytext);
               }
           }
      printf("%s is not a decimal number", yytext);
    }
  %%
  main()
   {
    printf("\n enter any number \n");
    yylex();
```

```
int yywrap()
  {
    return 1;
  }
```

Output

```
[ ]# lex  decimal.l
[ ]# cc  lex.yy.c
[ ]#  ./a.out
 Enter any number
  0.25
  0.25 is a decimal number
```