Chapter

1

# Introduction to Compiler

**Syllabus**

- Introduction to compilers
- Translator issues.
- Why to write compiler.
- Compilation process in brief.
- Front end and backend model.
- Compiler construction tools.
- Interpreter and the related issues.
- Cross compiler.
- Incremental compiler.
- Boot strapping.
- Byte code compilers.

## 1.1 Translator Issues :

- The translator reads the source program written in one language and translates it to an equivalent program written in another language (which is a target language). If source language is High Level Language and Target Language is machine language then translator is called as Compiler. The output program i.e. machine language program is called as Object Program or module.
- This object module is a relocatable machine language form of source program.
- There are various types of translator like assembler compiler, interpreter etc.

**Fig. 1.1.1 : Translator**

- These translators are used as per the requirement of user. If source program written in assembly language need to translate in machine language assembler is used.
- A program loader performs the important functions of loading and link editing. The process of loading consist of relocation of the object modules which is the process of allocation of load time addresses i.e. actual physical address in the memory and placing the relocated instructions or code and data in memory at the proper locations.
- Loader is also responsible for the initiation of the program.
- The link editor links the object modules and allows us to make a single program from several files of relocatable object modules by resolving mutual references.
- These linked files may be library files provided by the system which may be referenced by any programmer.
- Thus execution of programming translator is **2 step process :**
  (i) Translation of source program and other referenced programs (if any).
  (ii) Loading of source program and other referenced program (if any) into main memory and initiation of execution of source program.

## 1.2 Why to Write Compiler ?

- A compiler translates a program written in high level language and generates assembly code or relocatable machine code. If compiler generates assembly program then assembler translates that assembly program to relocatable machine code.
- The target machine determines the instructions to be used in the target program.
- The machine code generated by the complier executes very efficiently.

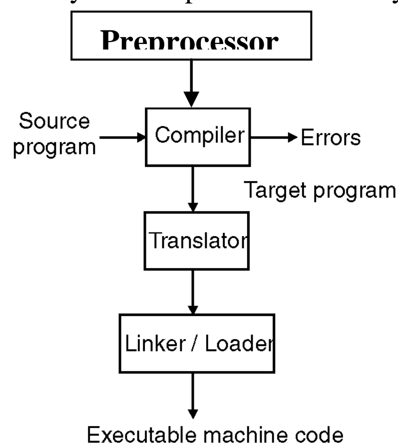



**Fig. 1.2.1 : Compiler**

- As compared to the simple one to one translation as used in assemblers, a compiler has to deal with many complex issues due to the nature of a high level programming language like.

**(1) Complex expressions and operator hierarchies :**

- ✧ In source program many arbitrarily large and complex expressions can exist.
- ✧ Hence the compiler must determine the correct evaluation order for the operators in the expressions.

  e.g. x + y * (p + q)
- ✧ For such expression compiler uses precedence of "*" and "+" operators.

**(2) Data-types :**

- ✧ The representation of the values in the execution of programs depends upon its data type. The compiler must ensure that the representation of the values is properly interpreted.
- ✧ For this, compiler keeps track of types of values and checks its compatibility with values being assigned to it.
- ✧ In case of non-compatibility of values compiler having type conversion feature automatically coverts the values as per compatibility.
- ✧ Compiler provides facility to define user defined data types.

**(3) Data structures :**

- ✧ Compiler provides storage mapping to access parts of data structure. This mapping must utilize knowledge about the allocation of memory for the data.
- ✧ The nature and the complexity of the mappings depend, on the nature of the data.

**Example :** Access value in array with x[8] and also value in struct employee like employee name.

**(4) Scope rules :**

- ✧ Due to block structure, a source program may consist of different name spaces in a program.
- ✧ The compiler must resolve the occurrence of each variable name in a program to determine which specific data item/data structure is represented by it.

**(5) Control structures :**

- ✧ These functions help in checking the validity of its use in a program.
- ✧ Compiler makes provision to vary value of loop controlled variable as specified in looping statement.

**(6) Optimization :**

- ✧ It is compiler performed rearrangement of the computation in a program such that the transferred program is semantically equivalent to the original program but executes faster.

**Ex. :** elimination of common sub expressions.

Very few computer professionals write even one compiler in their entire professional career. So why should all learn compiler construction? Because the principles of compilation are useful in many non-compilation environments, e.g.

1. **Interfaces and command language :** The interfaces to most software packages involves the use of a command language. This language needs to be compiled or interpreted.
2. **Language migration :** Automated tools need to be developed to migrate the software written using old language.
3. **Re-engineering :** Automated tools need to be developed to "Rewrite" the software developed using older design methodologies so as to utilise the current design methodologies.

## 1.3 Compilation Process :

**äää [ University Exam : May 2006 !!! ]**

- The fundamental language processing model for compilation consists of two steps processing of a source program.
  (1) Analysis phase of source program.
  (2) Synthesis phase of a source program.
- 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.

### 1.3.1 The Analysis Step Consists of Three Phases :

**(1) Linear analysis or lexical analysis :**

- In Lexical Analysis input characters of source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning.
- Token is the smallest meaningful entity of program are produced as output.

**(2) Hierarchical analysis or Syntax Analysis :**

- Syntax analysis determines the structure of source program string by grouping the tokens into tree (hierarchical structure) in which each node represent an operation

and children of a node represent arguments of the operation.

**(3) Semantic analysis :**

- Semantic analysis performed certain checks

  **Ex:**

  ✧ Whether operator have compatible arguments or not

  ✧ Finding out meaning of a source string

  ✧ Ensuring that components of program fit together meaningfully.

### 1.3.2 Synthesis of a Source Program :

- After three phases compiler generate an explicit intermediate representation as a program for an abstract machine.
- The intermediate code should be easy to produce and easy to translate into the target program.
- Synthesis phase consists of :

  **(1) Code optimization :** This phase of synthesis step improve the intermediate code so that faster execution of machine code can be produced.

  **(2) Code generation :** This phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.

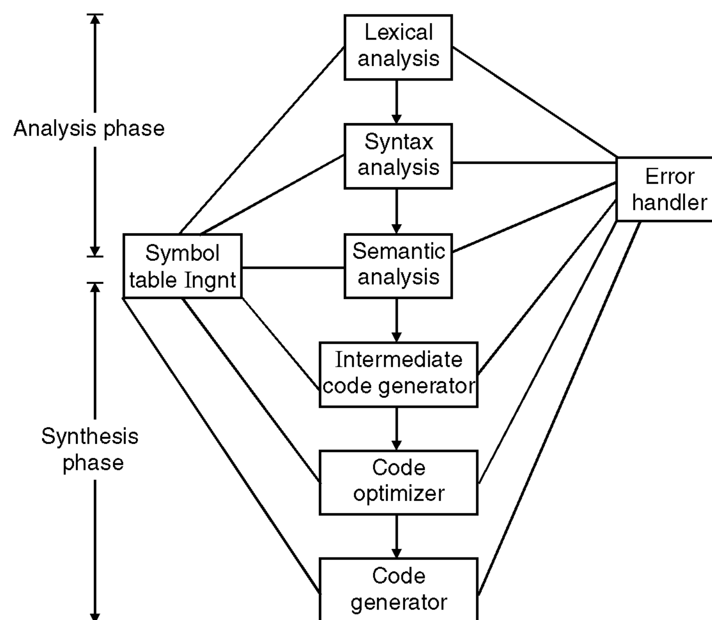



**Fig. 1.3.1 : Process of compilation**

### 1.3.3 Compilation Process with Example :

**äää [ University Exam : Dec. 2004, May 2005 !!! ]**

#### (1) Lexical Analysis :

- In compilation process lexical analysis is also called as linear analysis or scanning.
- A program that performs Lexical Analysis is also called as Lexical Analyzer or Scanner.
- For example we consider following expression for lexical analysis to code generation in compilation process.

$$\mathbf{P = Q + R * 3}$$

- From the expression lexical analysis breaks the source string to build uniform descriptions and groups into the following types of tokens. Like P, Q, R are identifiers and generates tokens id1, id2, id3 respectively and make the entry of identifier in symbol table.
- For given expression, lexical analyzer generates following tokens.
  Assignment operator =
  Arithmetic operator +
  Arithmetic operator *
  constant 3
- Blank characters from whole statement are eliminated by lexical analysis.

#### (2) Syntax Analysis :

- It is also called as parsing and is done by the program which is known as Parser or Syntax analyzer.
- The groups of tokens which are separated by lexical analysis into grammatical phrases, which are used by further phases of compiler.
- Syntax analyzer represents its output in the form of parses tree. Like :

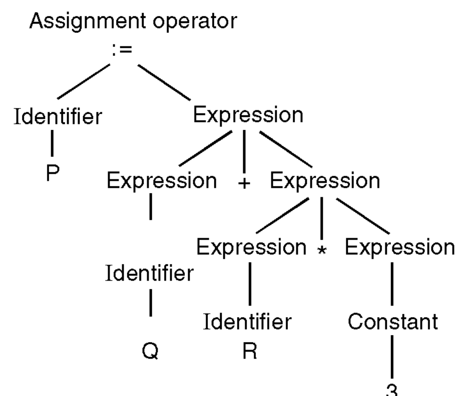



**Fig. 1.3.2 : Parse tree for P : = Q + R * 3**

- This parse tree is hierarchical structure as the expression having different operations to solve.
- Parse tree is used by remaining phases with priorities of the operation. Like R ∗ 3 multiplication performed before Q + R.
- The syntax tree, Fig. 1.3.3, describes the syntactic structure of the input.
- A more common internal representation of this syntactic structure is given by the parse tree in Fig. 1.3.2.

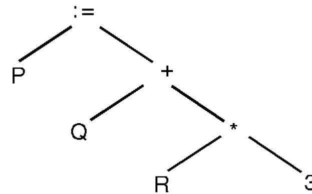



**Fig. 1.3.3 : Syntax tree for P : = Q + R * 3**

- A syntax tree is compressed representation of the parse tree in which operators appear as the interior nodes and the operands of an operator are the children of the node for that operator.

**(3) Semantic Analysis :**

- The semantic analysis phase uses parse tree of the source program to check the semantic errors and to find out the meaning of statement (identify operators and operand of expression) and collect relative information for further phases.
- Semantic analysis performs major task of type checking. In this phase compiler checks that operands used in expression are as per the specification given in source language.
- For example,

$$P = Q + R * 3$$

Where R is real and 3 is integer compiler takes care of this kind of type conversion and converts integer to real. i.e. constant 3 converted to real as 3.0.

**(4) Intermediate Code Generation :**

- After checking of syntax and semantic errors compiler generates internal representation of the source program.
- This internal representation should have important properties as
  - ✧ It should be easy to produce.
  - ✧ It should be easy to translate into the target program. This improves the efficiency of the compiler.
- Generated representation can have one of various forms like triple, quadruple.

- Triple also called as "three-address code" which consists of sequence of instructions each of which as at most three operands. For example,

$$t1 = \text{inttoreal}(3)$$
$$t2 = id3 * t1$$
$$t3 = id2 + t2$$
$$id1 = t3$$

- t1, t2, t3 are temporary names used to hold computed values.
- To generate these instructions compiler has to decide order in which operations is to be performed.

**(5) Code Optimisation :**

- The code optimization phase attempts to improve the intermediate code, so that faster running machine code can be produced.
- Intermediate code generation phase generates the code as per the tree representation, even though there is a better way to perform the same calculation using few instructions.
- For example :

$$t1 = R * 3$$
$$id1 = id2 + t1$$

- As inttoreal (3) is already converted by semantic analysis and intermediate code generated for the same, there is no need to convert inttoreal (3) every time of compilation. Also t3 only used in intermediate code to store value in id1 which is get eliminated and value is directly stored in id1 which reduces the time of execution of the machine code.
- This phase sometimes placed after code generation, which optimizes generated machine code.

**(6) Code Generation :**

- The last phase of the compiler is generation of target code (machine code). In this phase intermediate (optimized) code is translated into a sequence of machine instructions that perform the same operation.
- For every variable memory locations (registers) are selected, and code is generated with reference to that memory locations.
- For example, register 1 and 2 are used.

MOVF id3, R2

MULF # 3.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

- In the generated instruction first and second operands specify source and destination respectively. F in each instruction tells us that instruction deal with floating point number.

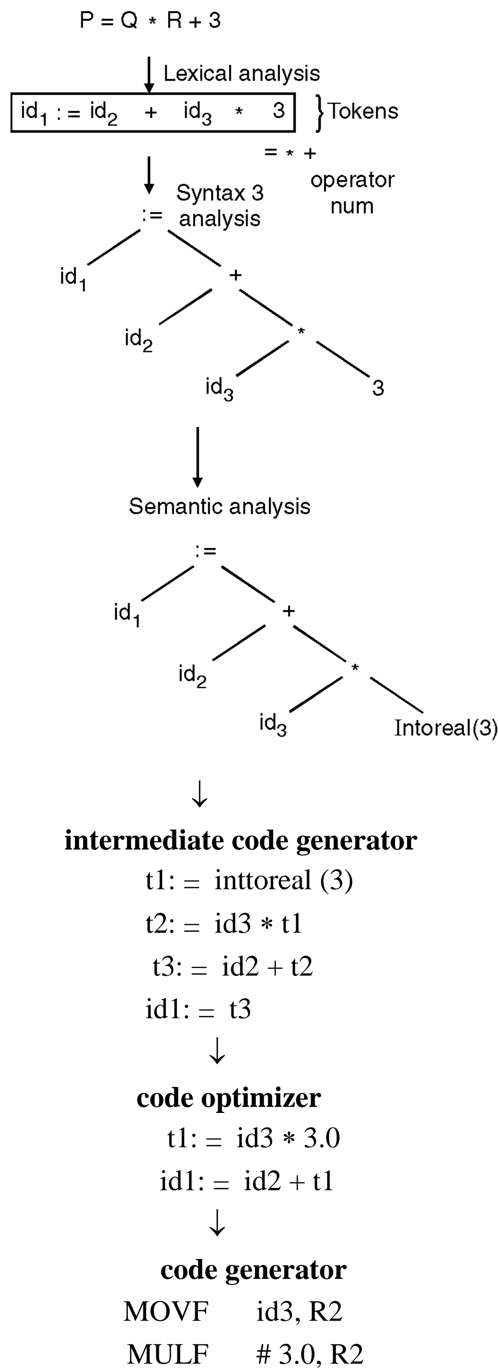
**(7) Symbol Table Management :**

- An essential function of compiler is to make record of identifier used in the source program with their various attributes. These attributes provide information about the storage allocated for identifier, its type, its scope, etc.
- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of identifier. This data structure helps to find the record of each identifier quickly and retrieve the data quickly.
- As the identifier detected by lexical analyzer it is entered into symbol table. All the attributes are not entered by lexical analysis.
- For Example : P, Q, R declared as variable are stored in symbol table by the lexical analyzer, and remaining phases enters information about identifier for further use.

**(8) Error Detection and Reporting :**

- The primary goal of a compiler is to generate code for programs. But programmers are humans and they make mistakes. As a result they frequently write incorrect programs.
- Therefore, an important goal of a compiler is to assist the programmer in detecting and locating errors.
- Error handing should be integrated with the compilation process.
- It has various goals :
  (a) It should detect errors in various phases of compiler.
  (b) It should report the errors clearly providing information about the cause and nature of errors so that the user can manage to correct the program.
  (c) It should recover from errors in order to continue processing and error detection for the rest of the program/statements.
- The syntax and semantic analysis phases usually handle large fraction of the errors detectable by the compiler.
- **Lexical errors** can be due to spelling errors as due to illegal characters in the source code which does not form any token of the language.

- **Syntax errors** occur due to errors in structures, missing operands and keywords, unbalanced parenthesis etc.
- **Semantic errors** are due to violation of rules of accessibility like use undefined values, incompatible operand.
- Fig. 1.3.4 shows phase wise translation of P = Q + R * 3



```
MOVF   id2, R1
ADDF   R2, R1
MOVF   R1, id1.
```

**Fig. 1.3.4. : Phase wise translation of P = Q + R * 3**

## 1.4 Front End and Back End :

- The phases of compiler are grouped into front end and back end. **This is logical organization of compiler.**
- The front end of the compiler includes those phases that are dependent on the source language and are independent of the target machine.
- Front end of the compiler normally includes lexical, syntactic, semantic analysis part. A certain amount of code optimization can be done by front end. The front end also includes the error handling that goes along with each of these phases.
- The back end includes those phases of the compiler that depend on the target machine, and generally those portions of the compiler that do not depend on the source language but the intermediate language.
- In the back end we find aspects of the code optimization phase and code generations along with the necessary error handling and symbol table operations.
- **The advantages of front end-back end organization :**
  (1) Take the front end of a compiler and change its associated back end to produce a compiler for the same source language and a different machine language.
  (2) Take the common back end for the different front ends to compile several different language into the same machine language.

## 1.5 Compiler Construction Tools :

**äää [ University Exam : Dec. 2003 !!! ]**

- The compiler writing is like a programming probably using software tools such as debuggers, profilers so on. Other more specialized tools have been developed to help implementation of various phases of a compiler.
- These systems are often referred to as compiler–compilers, compiler generators or translator–writing systems. They are oriented around a particular model of languages. They are most suitable for generating compilers of language similar to the model.
- Some general tools created for the automatic design of specific compiler components. In this, 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.

- Some useful compiler construction tools:
  1. **Parser Generators**: This tool produces syntax analyzer using context free grammar as input. The syntax analysis phase is not easy to implement. Many parser generators utilize powerful parsing algorithm that are too complex to be carried out by hand.
  2. **Scanner Generators:** These tools automatically generate lexical analyzers using regular expressions as input. The basic organization of the resulting lexical analyzer is effect of a finite automation. Because RE (Regular Expression) which can be converted into Finite Automata.
  3. **Syntax-directed Translation Engines:** These tools produce collection of routines that walk the parse tree generating intermediate code. The basic idea is that one or more translation is defined in terms nodes in the tree.
  4. **Automatic Code Generators:** This tool accepts a collection of rules. These rules define the translation of each operation of the intermediate language into the machine language for the target machine.
  5. **Data-flow engines:** Much of the information needed to perform good code optimization involves "data flow analysis" which is the process of gathering of information about how values are transmitted from one part of a program to each other part.

## 1.6 Interpreter :

- Interpreter is one of translator which has important advantages over compilation. In interpreter, the data and the source program are input to the interpreter instead of producing any object module as in the compilation, the interpreter directly produces the results by performing the operations of the source program on its data in Fig. 1.6.1.

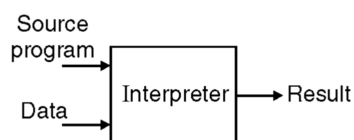



**Fig. 1.6.1**

- A major disadvantage is that a source program needs to be interpreted whenever it has to be executed. Interpreter is less efficient than compiler as for every execution program has to be interpreted.
- But interpreter have certain advantage over compiler as :
  (1) It can handle certain language features which cannot be compiled e.g. languages like APL are normally interpreted as it involves features about the data such as the size and shape of arrays which cannot be deducted at compile time.
  (2) Interpreter can be made portable since it does not produce machine language programs. It saves time required for assembling and linking a real program. An

interpreter gives as an improved debugging environment because it can check for errors like out-of-bounds array indexing at run time.

(3) Interpreters are used to interpret programs in language as HTML.

**There are some questions about interpreter :**

(1) Similarities and difference between interpreter and compiler

(2) Situation in which interpreter preferred over compiler

- **Similarities and difference between interpreter and compiler with merits and demerits :**

1. Execution of a compiled state involves execution of few machine instructions, while its interpretation would involve execution of hundreds of instructions.
2. Execution of the program using interpreter in one step process whereas execution of the program is two step process as
   a. Compilation of the program and preparation of the object program
   b. Loading of the Object Program in Main Memory and then execution of the program.

Using above information, we can argue that interpreter are superior to compiler in a program development environment. Characteristics of Program development environment are as follows :

**(1)** During program development, a user is typically testing and debugging program by

- Running the program on known data to evaluate its corrections.
- Editing the program to remove logical errors.

**(2)** Test data is selected so as to restrict the amount of execution of the program. If a program consists of say 300 state the number of state actually visited during the test run would be typically fewer than 300.

**(3)** Every test run is typically followed by some modification to the program. Hence if a compiler is being used the program will have to be compiled for every test run.

### 1.6.1 Situation in which Interpreter Preferred Over Compiler :

- Thus interpretation is cheaper in the case of the CPU time cost, interpreters are well suited for a program development environment, but not for a production environment.
- The interpreters are simple to develop than compilers because compilers have to generate target machine code which is a complex task, while interpreters do not have to perform code generation. Interpreters are also useful in environments where interpreter may be smallest in size than a compiler. It can be used in environments where only one execution is desired e.g. database or operating system commands.

## 1.7 Cross-Compiler :

- A compiler may run on one machine and produce target code for another machine, such a compiler is often called as cross-compiler.
- Suppose we write a cross-compiler for a new language L in implementation language S to generate code for machine N, that is we create $L_SN$.
- If an existing compiler for S runs on machine M and generate code for M, it is characterised by $S_MM$. If we run $L_SN$ through $S_MM$, we a compiler $L_MN$ that is a compiler from L to N that runs on M.

  By putting this process together the T-diagrams for these compilers in Fig. 1.7.1

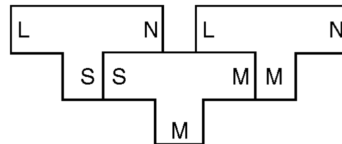


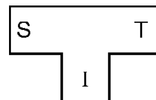

**Fig. 1.7.1**

- T-diagram Fig. 1.9.1 can be thought of as an equation :

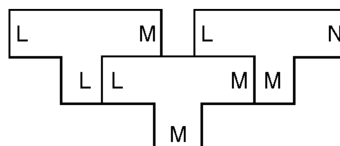
$$L_SN + S_MM = L_MN$$

## 1.8 Boot-strapping :

- A compiler is complex enough program that we would like to write it in a friendlier language than assembly language. The facility offered by a language to compile itself is bootstrapping. e.g. In UNIX programming environment compilers are usually written in C. One form of bootstrapping builds up a compiler for larger and larger subsets of language. For bootstrapping purpose a compiler is characterized by three languages : the source language **S** that it compiles, the target language **T** that it generates code for and the implementation language I that it is written in. Represent the three language using T-diagram.

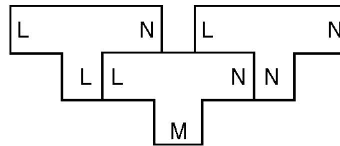



- Advantages of bootstrapping can be realized fully by writing a compiler in the language, which is compiled by compiler (i.e. the language translated by compiler )
- e.g. :




- We write a compiler $L_LN$ for language L in L to generate code for machine N. Development takes place on a machine M, where an existing compiler $L_MM$. We obtain

a cross-compiler here $L_MN$ that runs on M, but produces code for N. The compiler $L_LN$ can be compiled a second time, this time using the generated cross compiler.




- With the result of bootstrapping and cross-compiler, $L_MN$ that runs on N and generate code for N.

## 1.9 Byte Code Compiler :

- A byte code compiler translates a complex high level language like LISP into a very simple language that can be interpreted by a very fast byte code interpreter or virtual machine.
- The internal representation of the simple language is a string of bytes hence the name **byte code compiler**.
- The compilation process eliminates a number of costly operations of the interpreter as variable lookup and setting up exists of nonlocal transfer of control. It also performs some inlining and makes the language tail-recursive. This means that tail calls are compiled as jumps and therefore iterations can be implemented using recursion.
- In any high level language for scientific and statistical computing there will always be a need to implement some code in a lower level language like to obtain acceptable performance.
- But developing code in higher language itself is usually much faster and the resulting code is typically easier to understand and maintain. But it is advantageous to allow use of a low level language as far down as possible. Byte code compilation helps significantly in this respect.
- It is possible to translate the byte code produced by the compiler into C code, which can then be compiled with a native a compiler and linked into the system dynamically or statically.
- The native approach does indeed produce additional speed up by factor of two by eliminating the byte code interpretation component, but significant cost is paid as the resulting object code is huge. As a result, this option has not been pursued.
- However a more sophisticated approach based on a higher level intermediate representation appears to be promising and will be considered as part of the development of a new virtual machine.

## 1.10 Incremental Compiler :

äää [ University Exam : Dec. 2003 !!! ]

- In compiler design there is often a trade off between compilation time and target code quality. This trade off is even more dedicate in an intermental compiler, where low response time s crucial.
- A source file and an object file are organized into logical blocks. An intermediate file is generated which stores information about the logical blocks in both the source file and the object file and their relationship with each other.
- Boundaries are established in the source program to define logical block and each block is termed as function. Each function is divided into a global region and local region.
- If changes are made in a particular local region in source file then only that region is recompiled and recompiled portion of the object file is patched into the object file by replacing the previous object code corresponding to that region.
- Significant time saving is realized by incrementally compiling only local regions of the source program changed are recompiled and patching it into the existing object file.

## 1.11 Pass of the Compiler :

äää [ University Exam : May 2005 !!! ]

- Pass is the process of the scanning and processing of the program at once.
- One pass can be considered as the collection of various phased of the compiler.
- Ex. In two pass compiler first pass may have Lexical, Syntax and Semantic Analysis and Intermediate code generation phases.
- Where as in the second pass of the compiler code optimization and code generation phases
- No of passes of the compiler is decided by following factors
  1. **Storage Limitation**.: Multi-Pass compiler can be executed by having oly one pass the main memory with additional overhead due to storage of output of one pass as it is consumed by another pass.
  2. **Forward References**: Use of a variable before its declaration is if allowed by Language then these issue can solved by using multipass compiler.
- **Optimization** : By storing output of one pass in some intermediate file we can apply some optimization techniques to it before submitting to next pass as input.

**Review Questions**

Q. 1 Draw a neat diagram showing various phases of compiler. Explain in brief each of these phase.

Q. 2 What do you understand by the term 'Finite State Automata'? differentiate between NFA and DFA. What is the role of finite state automata in compiler construction ?

Q. 3 Discuss the merits and demerits of a compiler and an interpreter.

Q. 4 What is front end and back end of compiler.

Q. 5 For the following statement in 'C' write the output of each phase of a compiler.

Do
  a = a – b
While (a < b && a> = 4/b – 10)

Q. 6 For the statement

A = b + c ∗ 60;

Write all the compilation phases with input and output to each phase.

Q. 7 Discuss the various compiler construction tools.

Q. 8 Describe in detail, the steps involved in construction of a DFA form a NFA.

## 1.12 University Questions and Answers :

**May 2003 – Total Marks 10**

**Q. 1** With the help of the block diagram explain phases of the compiler. Also write down output of each phase of the compiler for expression a:=b+c*60. **(Section 1.3)** **(10 Marks)**

**Dec 2003 – Total Marks 12**

**Q. 2** What is Incremental Compiler? What are basic features of incremental compiler ? **(Section 1.10)** **(6 Marks)**

**Q. 3** With the help of block schematic explain how compiler-compiler can reduce the effort in implementing new compiler ? **(Section 1.5)** **(6 Marks)**

**May 2005 – Total Marks 4**

**Q. 4** Define the concept of the pass of the compiler? Which factors decides number of the passes for the compiler ? **(Section 1.11)** **(4 Marks)**

**May 2006 – Total Marks 14**

**Q. 5** With the help of the block diagram explain phases of the compiler. Also write down output of each phase of the compiler for expression P=I+R*80. **(Section 1.3)** **(10 Marks)**

**Q. 6** Define Pass of the compiler **(Section 1.11)** **(4 Marks)**

❑❑❑