1 Introduction

1.1 Introduction

Compilers are basically translators. Designing a compiler for some language is a complex and time consuming process. Since new tools for writing the compilers are available this process has now become a sophisticated activity. While studying the subject compiler it is necessary to understand what is compiler and how the process of compilation can be carried out. In this chapter we will get introduced with the basic concepts of compiler. We will see how the source program is compiled with the help of various phases of compiler. Lastly we will get introduced with various compiler construction tools.

1.2 Translator

A translator is one kind of program that takes one form of program as input and converts it into another form. The input program is called **source language** and the output program is called **target language**. The source language can be low level language like assembly language or a high level language like C, C++, FORTRAN.

The target language can be a low level language or a machine language.

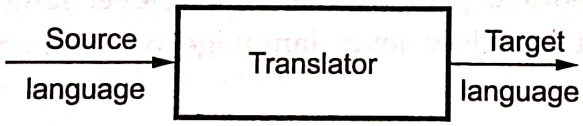


Fig. 1.1 Translator

Types of Translator

There are two types of translators **compiler** and **assembler.**

Basic Functions of Translator

1. The translator is used to convert one form of program to another.
2. The translator should convert the source program to a target machine code in such a way that the generated target code should be easy to understand.
3. The translator should preserve the meaning of the source code.
4. The translator should report errors that occur during compilation to its users.
5. The translation must be done efficiently.

1.3 What is Compiler ?

In this section we will discuss two things : ***"What is compiler ?*** And ***"Why to write compiler ?"*** Let us start with *"What is compiler ?"*

> Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program).

During this process of translation if some errors are encountered then compiler displays them as error messages. The basic model of compiler can be represented as follows.

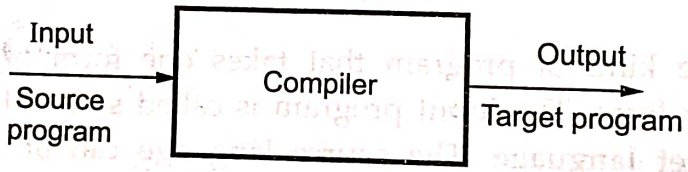


Fig. 1.2 Compiler

The compiler takes a source program as higher level languages such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

Key Point: *Compiler and assemblers are two translators. Translators convert one form of program into another form. Compiler converts high level language to machine level language while assembler converts assembly language program to machine level language.*

1.3.1 Compiler : Analysis-Synthesis Model

PU : Dec.-2008

The compilation can be done in two parts : analysis and synthesis. In **analysis part** the source program is read and broken down into constituent pieces. The syntax and the meaning of the source string is determined and then an intermediate code is created from the input source program. In **synthesis part** this intermediate form of the source language is taken and converted into an equivalent target program. During this process if certain code has to be optimized for efficient execution then the required code is optimized. The analysis and synthesis model is as shown in Fig. 1.3.

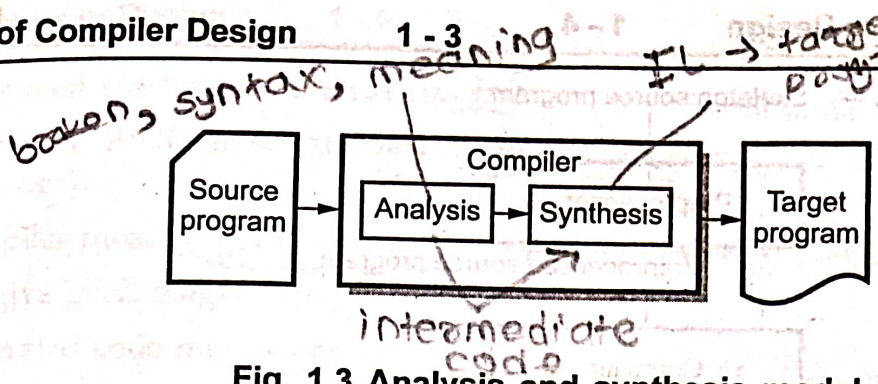


Fig. 1.3 Analysis and synthesis model

The analysis part is carried out in three sub parts

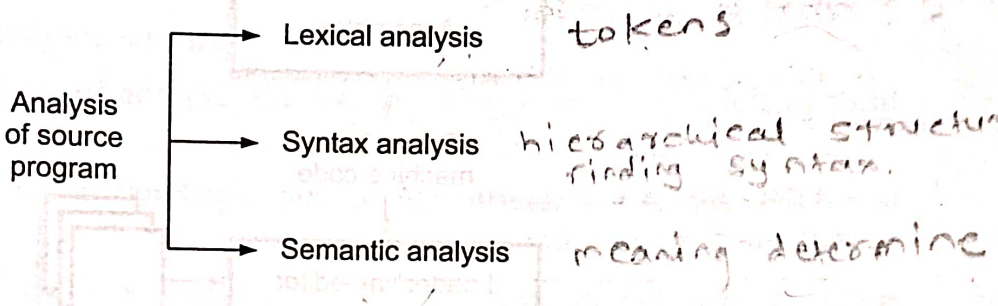


Fig. 1.4

1. **Lexical Analysis** - In this step the source program is read and then it is broken into stream of strings. Such strings are called **tokens.** Hence tokens are nothing but the collection of characters having some meaning.

2. **Syntax Analysis** - In this step the tokens are arranged in **hierarchical structure** that ultimately helps in finding the syntax of the source string.

3. **Semantic Analysis** - In this step the **meaning** of the source string is determined.

In all these analysis steps the meaning of the every source string should be unique. Hence actions in lexical, syntax and semantic analysis are uniquely defined for a given language. After carrying out the synthesis phase the program gets executed.

1.3.2 Execution of Program

To create an executable form of your source program only a compiler program is not sufficient. You may require several other programs to create an executable target program. After a synthesis phase a target code gets generated by the compiler. This target program generated by the compiler is processed further before it can be run which is as shown in the Fig. 1.5.

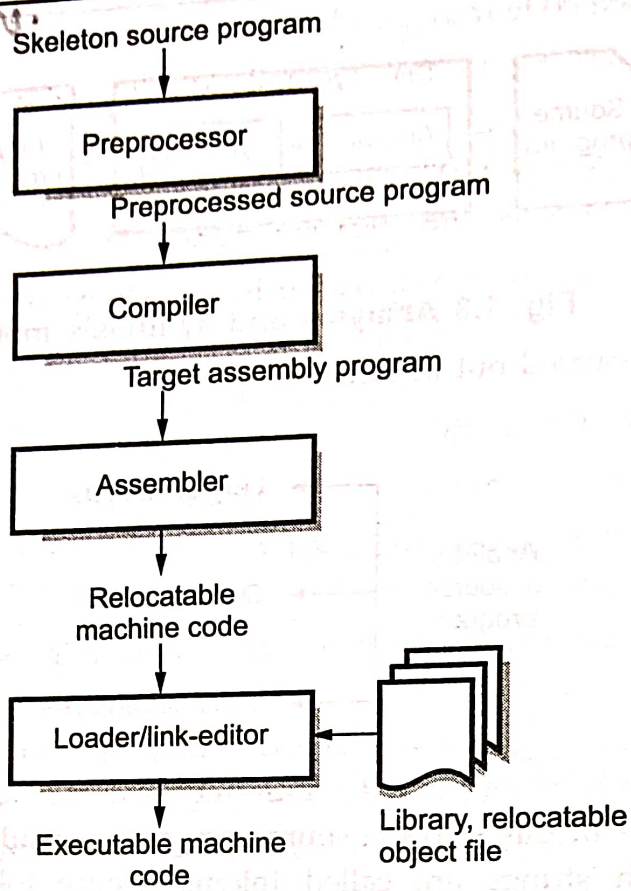


Fig. 1.5 (a) Process of execution of program

The compiler takes a source program written in high level language as an input and converts it into a target assembly language. The assembler then takes this target assembly code as input and produces a relocatable machine code as an output. Then a program **loader** is called for performing the task of **loading and link editing**. The task of loader is to perform the relocation of an object code. **Relocation** of an object code means allocation of load time addresses which exist in the memory and placement of load time addresses and data in memory at proper locations. The link editor links the object modules and prepares a single module from several files of relocatable object modules to **resolve** the **mutual references**. These files may be library files and these library files may be referred by any program.

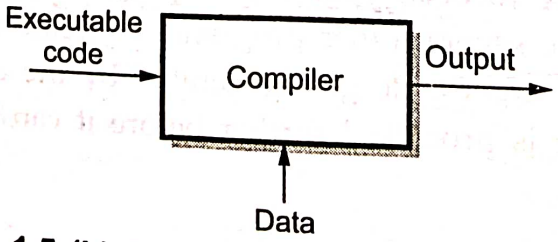


Fig. 1.5 (b) Process of execution of program

Properties of Compiler

When a compiler is built it should posses following properties.

1. The compiler itself must be bug-free.
2. It must generate correct machine code.

3. The generated machine code must run fast.
4. The compiler itself must run fast (compilation time must be proportional to program size).
5. The compiler must be portable (i.e. modular, supporting separate compilation).
6. It must give good diagnostics and error messages.
7. The generated code must work well with existing debuggers.
8. It must have consistent optimization.

1.3.3 Analysis of Source Program

The source program can be analysed in three phases -

1) **Linear Analysis :** In this type of analysis the source string is read from left to right and grouped into tokens.

   For example - Tokens for a language can be identifiers, constants, relational operators, keywords.

2) **Hierarchical Analysis :** In this analysis, characters or tokens are grouped hierarchically into nested collections for checking them syntactically.

3) **Semantic Analysis :** This kind of analysis ensures the correctness of meaning of the program.

1.3.4 Why should we Study Compilers ?

Following are some reasons behind the study of Compilers.-

1. The machine performance can be measured by amount of Compiled code.
2. Various programming tools such as debugging tools, source code validating tools can be developed for convinience of programmer.
3. The abilities of programming languages can be understood.
4. The study of compilers inspire to develop new programming languages that satisfies the need of programmer. The study of compilers helped to develop query languages, object oriented programming languages and visual programming languages.
5. Various system building tools such as LEX and YACC can be developed by analytical study of compilers.

1.4 Phases of Compiler

PU : Dec.-2008

As we have discussed earlier the process of compilation is carried out in two parts : **analysis and synthesis.** Again the analysis is carried out in three phases : lexical analysis, syntax analysis and semantic analysis. And the synthesis is carried out with the help of intermediate code generation, code generation and code optimization. Let us discuss these phases one by one.

1. Lexical Analysis

The lexical analysis is also called **scanning**. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of strings called token. A token is a sequence of characters having a collective meaning. For example if some assignment statement in your source code is as follows,

total = count + rate * 10

Then in lexical analysis phase this statement is broken up into series of tokens as follows.

1. The identifier **total**
2. The **assignment** symbol
3. The identifier **count**
4. The **plus** sign
5. The identifier **rate**
6. The **multiplication** sign
7. The constant number **10**

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

2. Syntax Analysis

The syntax analysis is also called **parsing**. In this phase the tokens generated by the lexical analyser are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens together. The hierarchical structure generated in this phase is called **parse tree** or **syntax tree**. For the expression total = count + rate *10 the parse tree can be generated as follows.

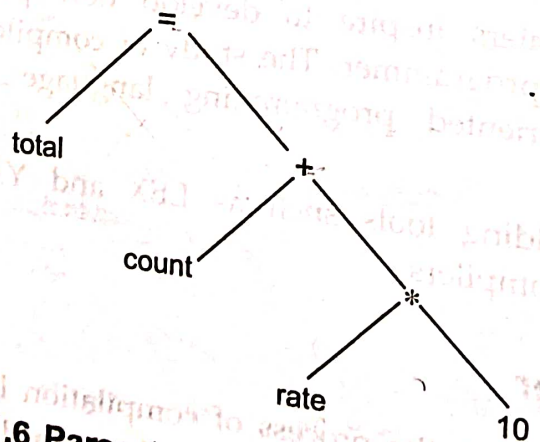


Fig. 1.6 Parse tree for total = count + rate *10

In the statement '*total = count + rate *10*' first of all *rate*10* will be considered because in arithmetic expression the multiplication operation should be performed before the addition. And then the addition operation will be considered. For building such type of

syntax tree the production rules are to be designed. The rules are usually expressed by context free grammar. For the above statement the production rules are -

1) E ← identifier
2) E ← number
3) E ← E1 + E2
4) E ← E1*E2
5) E ← (E)

where E stands for an expression.

- By rule (1) count and rate are expressions and
- by rule(2) 10 is also an expression.
- By rule (4) we get rate*10 as expression.
- And finally count +rate*10 is an expression.

3. Semantic Analysis

Once the syntax is checked in the syntax analyser phase the next phase i.e. the semantic analysis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if ...else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

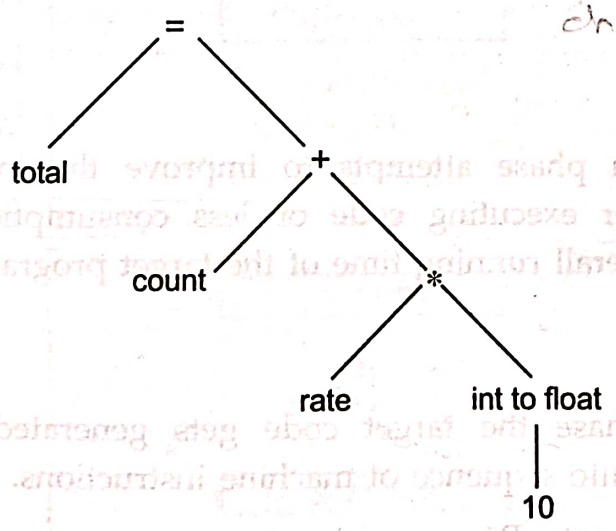


Fig. 1.7 Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

4. Intermediate Code Generation

The intermediate code is a kind of code which is easy to generate and this code can b easily converted to target code. This code is in variety of forms such as *three address cod quadruple, triple, posix*. Here we will consider an intermediate code in three address cod form. This is like an assembly language. The three address code consists of instruction each of which has at the most three operands. For example,

```
t1 := int to float (10)
t2 :=  rate × t1
t3 := count + t2
total := t3
```

There are certain **properties** which should be possessed by the three address code an those are,

1. Each three address instruction has at the most one operator in addition to th assignment. Thus the compiler has to decide the order of the operations devised by the three address code.
2. The compiler must generate a temporary name to hold the value computed by each instruction.
3. Some three address instructions may have fewer than three operands for example first and last instruction of above given three address code. i.e.

```
t1 := int to float (10)
total := t3
```

5. Code Optimization

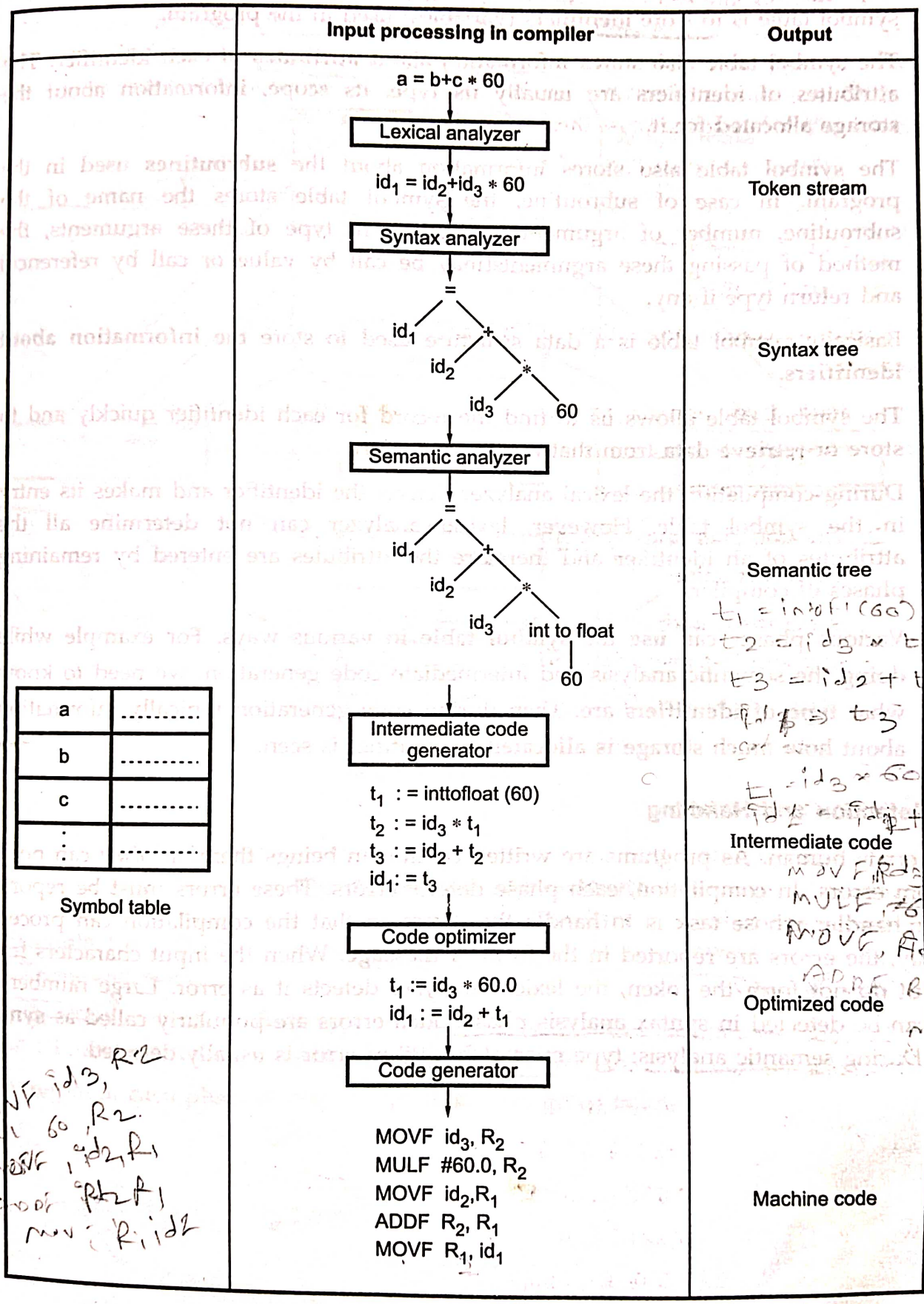
The code optimization phase attempts to improve the intermediate code. This i necessary to have a faster executing code or less consumption of memory. Thus b optimizing the code the overall running time of the target program can be improved.

6. Code Generation

In code generation phase the target code gets generated. The intermediate co instructions are translated into sequence of machine instructions.

```
MOV rate, R1
MUL #10.0, R1
MOV count, R2
ADD R2, R1
MOV R1, total
```

Example - Show how an input a = b+c *60 get processed in compiler. Show the output at each stage of compiler. Also show the contents of symbol table.



Symbol Table Management

- To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program.
- The symbol table also stores information about **attributes** of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it.
- The symbol table also stores information about the **subroutines** used in the program. In case of subroutine, the symbol table stores the name of the subroutine, number of arguments passed to it, type of these arguments, the method of passing these arguments(may be call by value or call by reference) and return type if any.
- Basically symbol table is a data structure used to store the **information about identifiers.**
- The symbol table allows us to find the record for each identifier quickly and to **store or retrieve data** from that record **efficiently.**
- During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of compiler.
- Various phases can **use** the **symbol table** in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what **type of identifiers** are. Then during code generation typically information about how much **storage is allocated** to identifier is seen.

Error Detection and Handling

To err is human. As programs are written by human beings therefore they can not be free from errors. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form of **message**. When the input characters from the input do not form the token, the lexical analyzer detects it as error. Large number of errors can be detected in syntax analysis phase. Such errors are popularly called as **syntax errors**. During semantic analysis; type mismatch kind of error is usually detected.

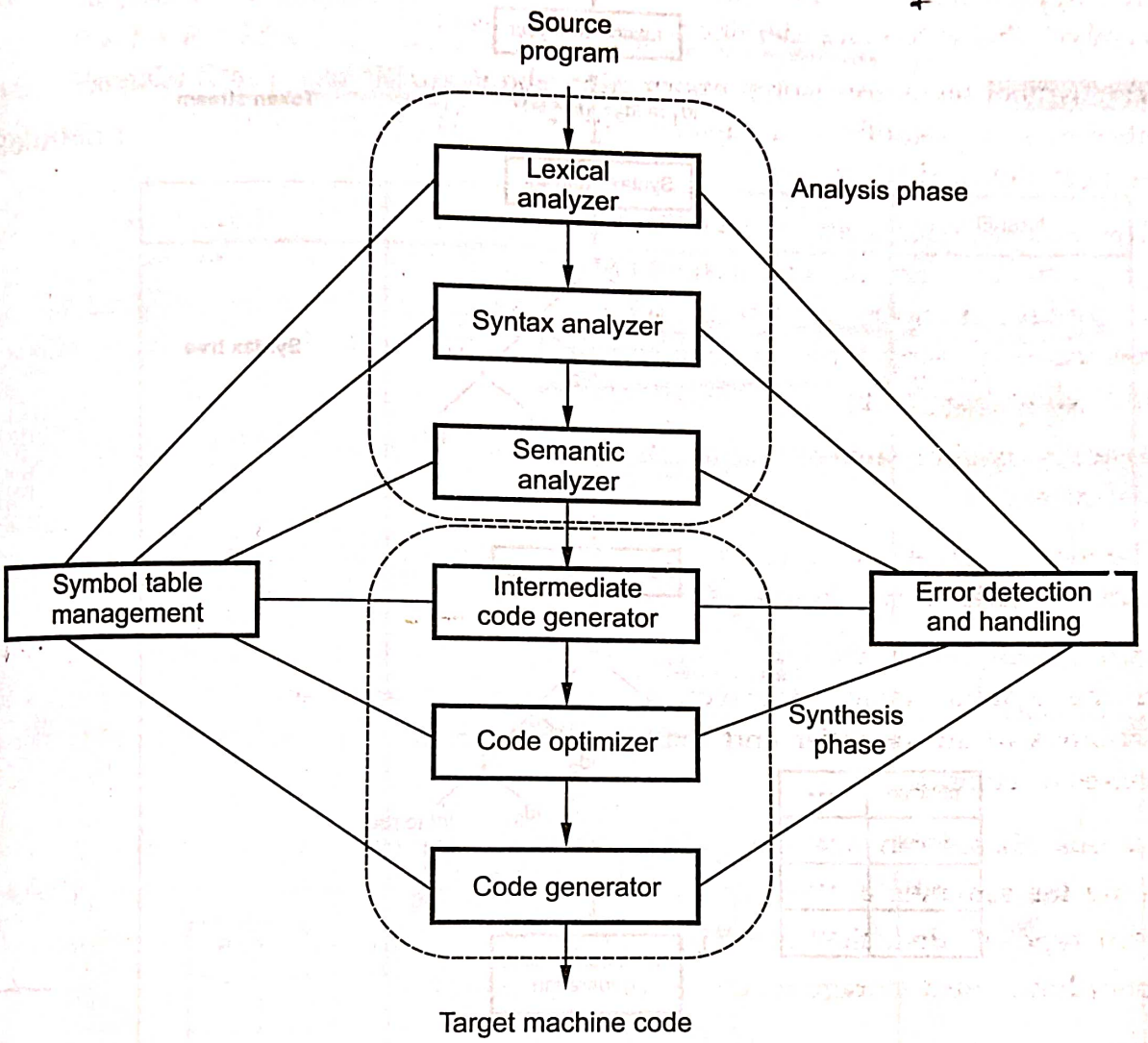


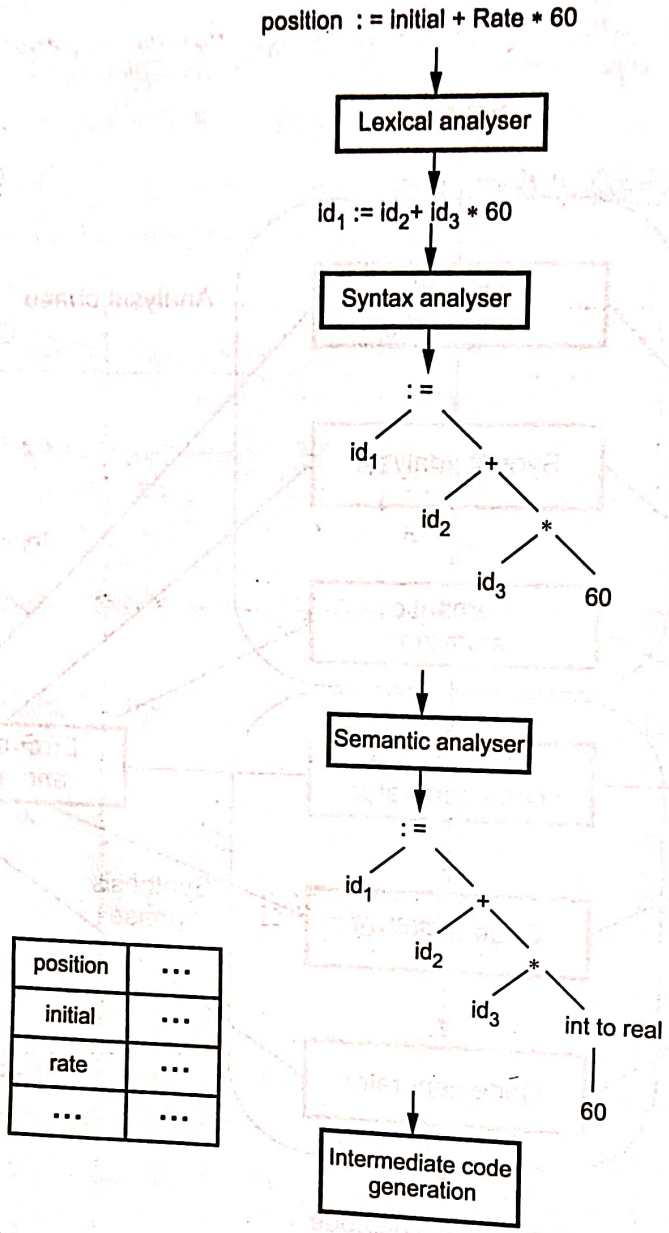
Fig. 1.8 Phases of compiler

Example 1.1 : *Describe the output for the various phases of compiler with respect to following statements :*

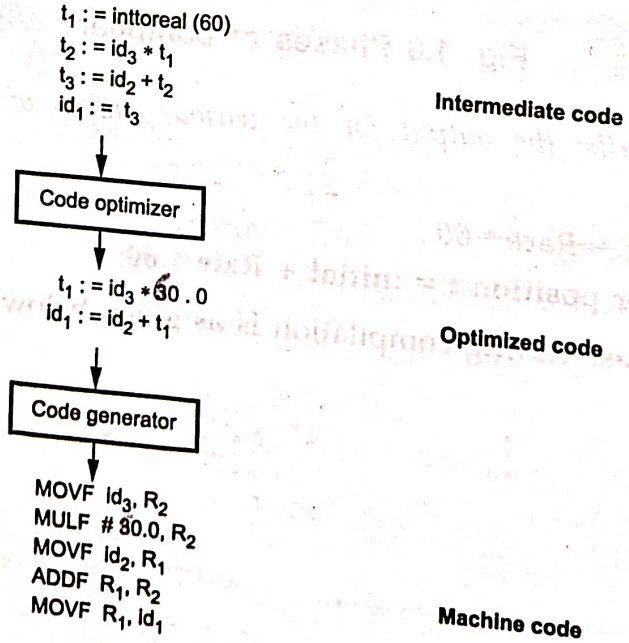
*position := initial + Rate * 60*

Solution : Phase result for position : = initial + Rate * 60

The output at each phase during compilation is as given below -



First operand represents source and second operand represents destination in the machine code.



Example 1.2 : *For the following source language statement, show the output at each of the phases of a compiler.*

$P = I + R * 23$

Variables 'P', 'I' and 'R' are of float type.

May-2007, 8 Marks

Solution :

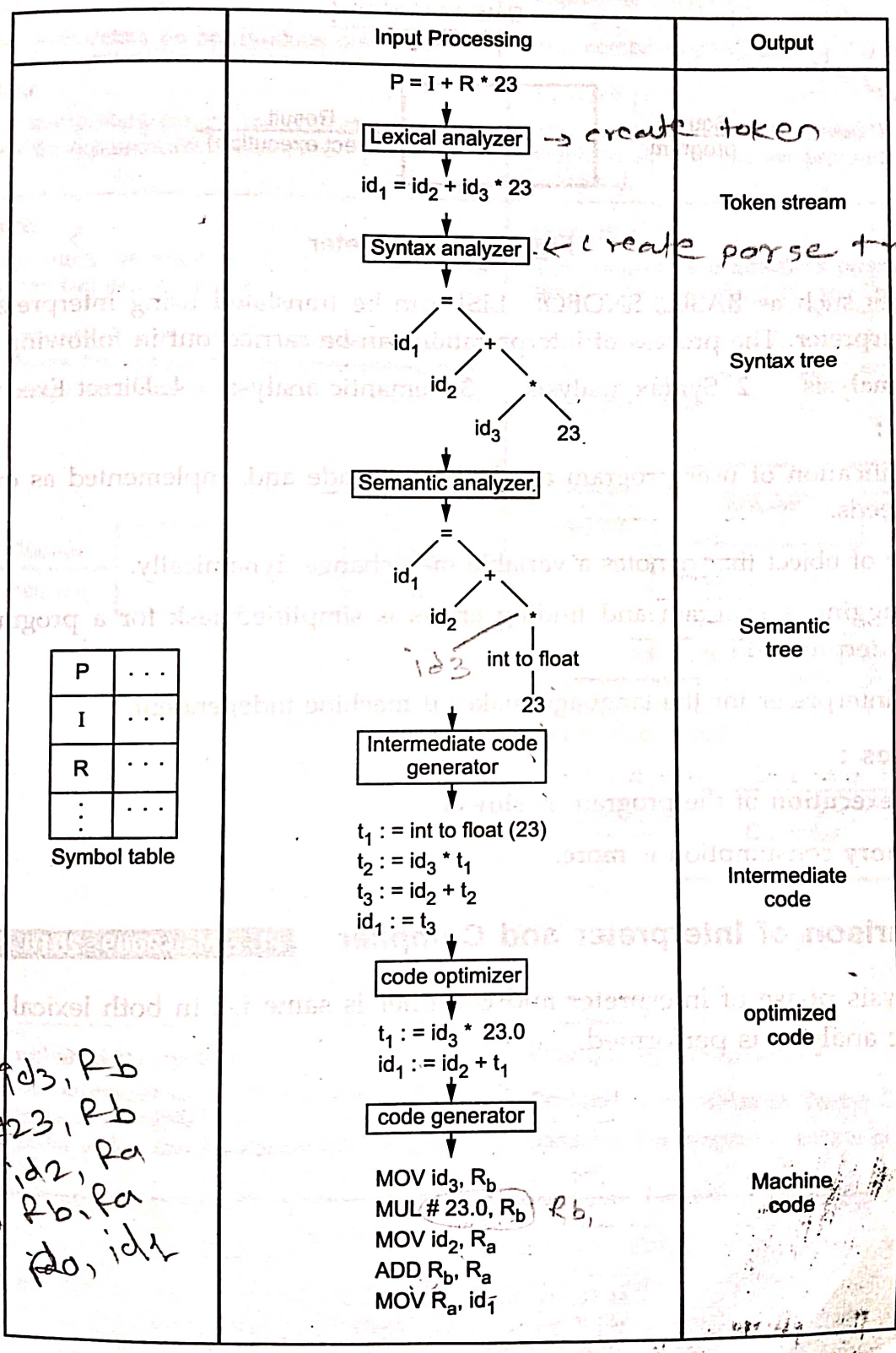


Fig. 1.9

1.5 Interpreter

PU : May-2007, 6 Marks

An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input. It does not produce the object code rather each time the program needs execution. The model for interpreter is as shown in Fig. 1.10.

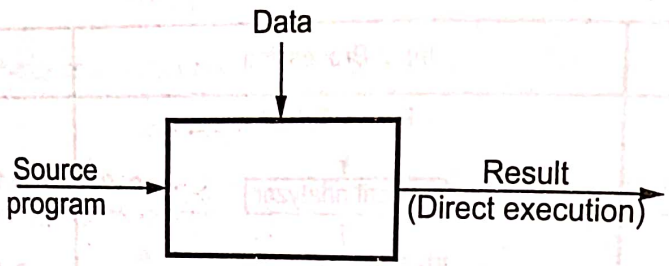


Fig. 1.10 Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis 2. Syntax analysis 3. Semantic analysis 4. Direct Execution

Advantages :

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a variable may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

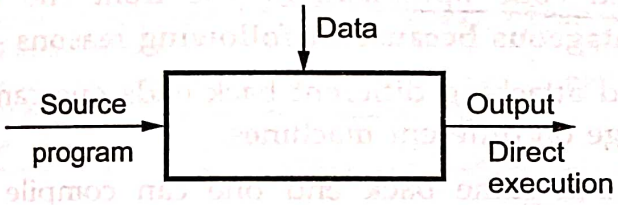
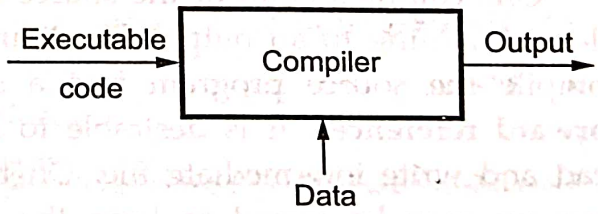
Disadvantages :

- The execution of the program is slower.
- Memory consumption is more.

1.6 Comparison of Interpreter and Compiler

PU : May-2009, 2010, 8 Marks

The analysis phase of interpreter and compiler is same i.e. in both lexical, syntactic and semantic analysis is performed.

Sr. No.	Interpreter	Compiler
1.	**Demerit :** The source program gets interpreted every time it is to be executed, and every time the source program is analyzed. Hence interpretation is **less efficient** than compiler.	**Merit :** In the process of compilation the program is analyzed only once and then the code is generated. Hence compiler is efficient than interpreter.
2.	The interpreters do not produce object code.	The compilers produce object code.
3.	**Merit :** The interpreters can be made **portal** because they do not produce object code.	**Demerit :** The compilers has to be present on the host machine when particular program needs to be compiled.
4.	**Merit :** Interpreters are **simpler** and give us **improved debugging environment.**	**Demerit :** The compiler is a **complex** program and it requires large amount of memory.
5.	An interpreter is a kind of translator which produces the results directly when the source language and data is given to it as input.  	An compiler is a kind of translator which takes only source program as input and converts it into object code  Then loader performs loading and link editing and prepares an executable code. Compiler takes this executable code and data as input and produces output. 
6.	**Examples of interpreter :** **A UPS Debugger** is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.	**Example of Compiler :** **Borland C compiler** or **Turbo C compiler** compiles the programs written in C or C^{++}.

PU : Dec.-2009, 4 Marks

1.7 Grouping of Phase

Front end back end

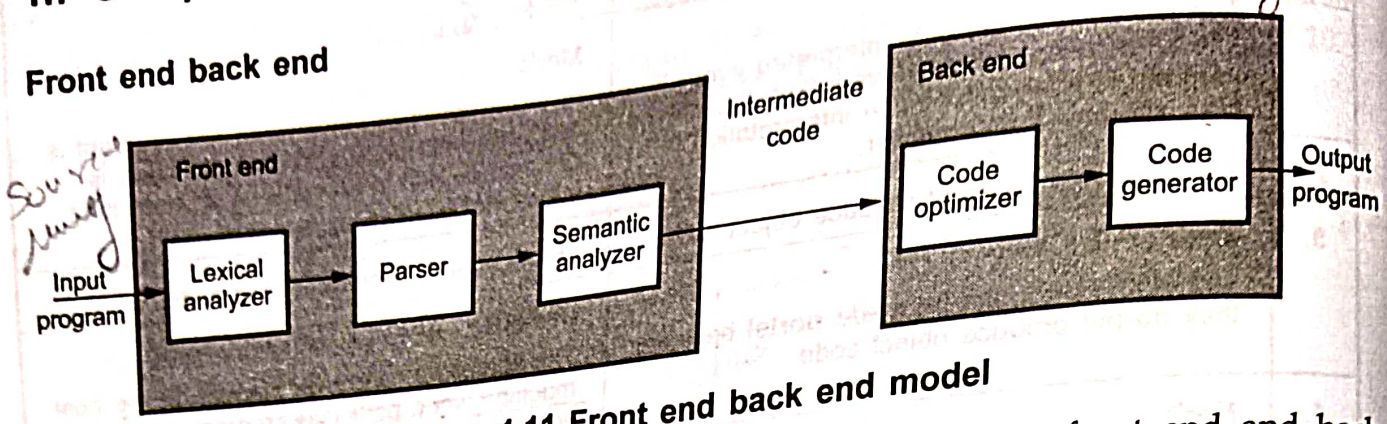


Fig. 1.11 Front end back end model

Different phases of compiler can be grouped together to form a front end and back end. The front end consists of those phases 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. It 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.

1.7.1 Concept of Pass

One complete scan of the source language is called **pass**. It includes reading an input file and writing to an output file. Many phases can be grouped one pass. It is difficult to compile the source program into a single pass, because the program may have some **forward references**. It is desirable to have relatively few passes, because it takes time to read and write intermediate file. On the other hand if we group several phases into one pass we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

In the first pass the source program is scanned completely and the generated output will be an easy to use form which is equivalent to the source program along with the additional information about the storage allocation. It is possible to leave a blank slots for missing information and fill the slot when the information becomes available. Hence there may be a requirement of more than one pass in the compilation process. A typical arrangement for optimizing compiler is one pass for scanning and parsing, one pass for semantic analysis and third pass for code generation and target code optimization. C and PASCAL permit one pass compilation, Modula-2 requires two passes.

1.7.1.1 Factors Affecting the Number of Passes

Various factors affecting the number of passes in compiler are -

1. Forward reference
2. Storage limitations
3. Optimization.

The compiler can require more than one passes to complete subsequent information.

1.7.2 Difference between Phase and Pass

PU. : Dec.-2007, 8 Marks

Phase	Pass
The processing of compilation is carried out in various steps. These steps are referred as phases. The phases of compilation are lexical analysis, syntax analysis, intermediate code generation, code generation and code optimization.	Various phases are logically grouped together to form a pass. The process of compilation can be carried out in single pass or in multiple passes.
The task of compilation is carried out in analysis and synthesis phase.	The task of compilation is carried out in single or multiple passes.
The phases namely lexical analysis, syntax analysis and semantic analysis are machine independant phases and the phases such as code generation and code optimization are machine dependant phases.	Due to execution of program in passes the compilation model can be viewed as front end and back end model.

1.8 Types of Compiler

In this section we will discuss various types of compilers.

1.8.1 Incremental Compiler

PU : Dec.-2007

Incremental compiler is a compiler which performs the recompilation of only modified source rather than compiling the whole source program.

The basic **features** of incremental compiler are,

1. It tracks the dependencies between output and the source program.
2. It produces the same result as full recompile.
3. It performs less task than the recompilation.
4. The process of incremental compilation is effective for maintenance.

PU : May-2008, 2 Marks

1.8.2 Cross Compiler

Basically there exists three types of languages

1. **Source language** i.e. the application program.
2. **Target language** in which machine code is written.
3. 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 compilation technique platform independency can be achieved.

To represent cross compiler T diagram is drawn as follows

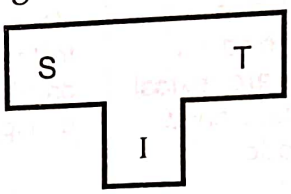


Fig. 1.12 (a) T diagram with S as source, T as target and I as implementation language

For source language L the target language N gets generated which runs on machine M.

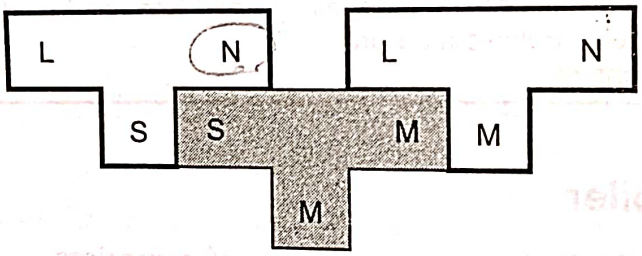


Fig. 1.12 (b) Cross compiler

For example :

For the first version of EQN compiler, the compiler is written in C and the command are generated for TROFF, which is as shown in Fig. 1.13.

The cross compiler for EQN can be obtained by running it on PDP-11 through C compiler, which produces output for PDP-11 as shown below -

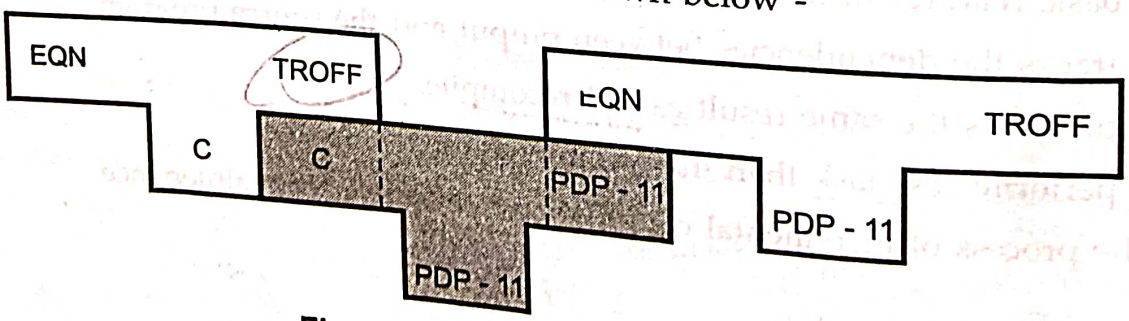


Fig. 1.13 Cross compiler for EQN

1.9 Bootstrapping

PU : Dec.-2007

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.

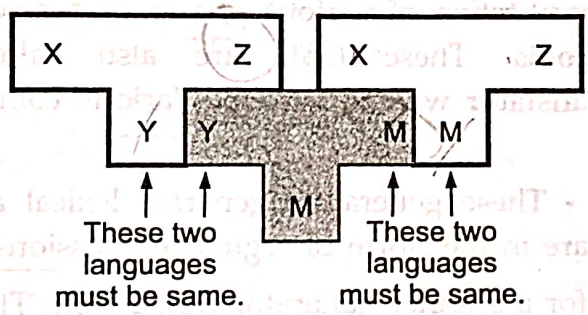
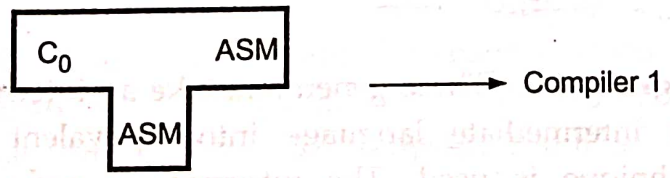


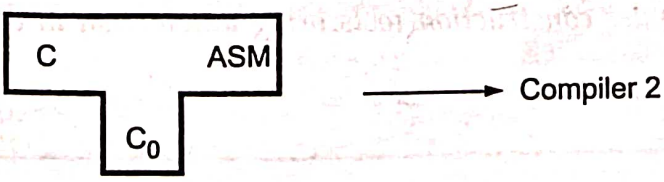
Fig. 1.14

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

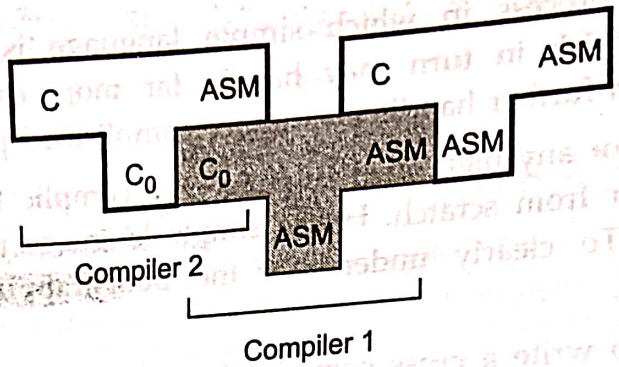
Step 1 : First we write a compiler for a small subset of C in assembly language.



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.

1.10 Compiler Construction Tools

PU : Dec.-2009, May-2010, 9 Marks

Writing a compiler is tedious and time consuming task. There are some specialized tools for helping in implementation of various phases of compilers. These tools are called **compiler construction tools**. These tools are also called as **compiler-compiler, compiler-generators or translator** writing system. Various compiler construction tools are given as below :

1. Scanner generator - These generators generate lexical analyzers. The specification given to these generators are in the form of regular expressions.

The UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

2. Parser generators - These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammar. Typically UNIX has a tool called YACC which is a parser generator.

3. Syntax-directed translation engines - In this tool the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.

4. Automatic code generator - These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language. The template matching technique is used. The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.

5. Data flow engines - The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

Key Point : *Compiler construction tools bring automation in compiler designing.*

1 Syntax 2 Parser generator 3 SDT 4 Automated code generator 5 Data flow engine

University Questions with Answers

Q.1 *Discuss the merits and demerits of an interpreter and a compiler.* **Dec.-2006, 5 Marks**

Ans. : Refer section 1.6

Q.2 *Draw a neat diagram showing various phases of a compiler. With a suitable example, write output of each phase. Discuss the advantages of designing a compiler with a front and back end.* **Dec.-2006; May-2008, 9 Marks**

Ans. : Refer sections 1.4 and 1.7.

Q.3 *With a neat diagram, explain various phases of a compiler. List various compiler writing tools you know.* **May-2007, 8 Marks**

Ans. : Refer sections 1.4 and 1.10.

Q.4 *For the following source language statement, show the output at each of the phases of a compiler.*

$P = I + R * 23$;

Variables 'P', 'I' and 'R' are of float type. **May-2007, 8 Marks**

Ans. : Refer example 1.2.

Q.5 *What is an interpreter ? Discuss its merits and demerits.* **May-2007, 6 Marks**

Ans. : Refer section 1.5.

Q.6 *What is difference between a phase and a pass of a compiler ? Explain machine dependent and machine independent phases of compiler.* **Dec.-2007, 8 Marks**

Ans. : Refer section 1.7.2.

Q.7 *Explain incremental, boot strapping, bytecode compilers with examples.* **Dec.-2007, 8 Marks**

Ans . : Refer sections 1.8.1 and 1.9.

Bytecode Compiler : A byte code compiler translates a complex high level language like LISP or Java into a very simple language that can be interpreted by a very fast byte code **interpreter** or **virtual machine**. The translated code is a simple stream of bytes. Hence the name is byte code compiler. The byte code program may be executed by directly parsing and executing instructions one at a time.

This type of compilers are also called as **dynamic translators** or **just in time (JIT)** Compiler. These type of compilers allow much better performance.

Q.8 *List and explain various compiler construction tools. List various operators used in BNF notation.* **Dec.-2007, May-2008, 8 Marks**

Ans. : Refer section 1.10.

Various operators used in BNF notation

The syntax of programming language constructs can be described by context free grammars or BNF (Backus-Naur-Form). A context free grammar has four components

1. Set of tokens, known as non terminals
2. Set of terminals
3. A set of productions where each production consists of a nonterminal called the left side of the production, an arrow and sequence of tokens and/or nonterminals, called the right side of the production.
4. A designation of one of the nonterminals as the start symbol.

For e.g.

$G = \{(A,B), (a,b), (S,P)\}$

$P \Rightarrow$ $S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Here (A,B) are nonterminals

(a,b) are terminals

$S \Rightarrow$ start symbol

$P \Rightarrow$ set of production rules

Q.9 *Explain the following :* **May-2008, 8 Marks**

i) Cross compiler *ii) Bootstrap compiler*

iii) Incremental compiler *iv) Compiler-compiler*

Ans. : i) Cross compiler - Refer section 1.8.2.

ii) Bootstrap compiler - Refer section 1.9.

iii) Incremental compiler - Refer section 1.8.1.

iv) Compiler-compiler - Refer section 1.10

Q.10 *What do you meant by analysis and synthesis of a source program ? What phases are involved in carrying out these tasks ? Explain in brief with suitable example source language statement or construct.* **Dec.-2008, 8 Marks**

Ans. : Refer sections 1.3.1 and 1.4.

Q.11 *List various phases of a compiler. For the following program statement in C, show the output of each phase of complier.*

$$P = I + R * 80$$

Assume that the type of variables P and R is float and I is an integer . The underline machine has only two registers say R_a and R_b and the machine provides instructions 'ADD' and 'SUB' for addition and multiplication of two operands respectively. Show the contents of symbol table also. **May-2009, 10 Marks**

Ans. : Refer example 1.2.

Q.12 *What is the difference between an interpreter and a compiler ?*

May-2009, 2010, 8 Marks

Ans. : Refer section 1.6.

Q.13 *Explain with example various compiler construction tools.*

May-2007, 2009, 2010; Dec.-2009, 9 Marks

Ans. : Refer section 1.10.

Q.14 *Why compilation phases are divided into front-end and back-end ? What are the advantages ?*

Dec.-2009, 4 Marks

Ans. : Refer section 1.7.

❑❑❑

Basics - Main - form