1 Overview of Compiling

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

JNTU : April/May-2005, 2007, 10 Marks

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.

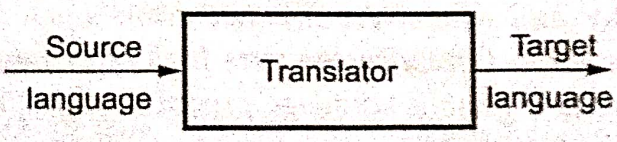


Fig. 1.1 Translator

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.

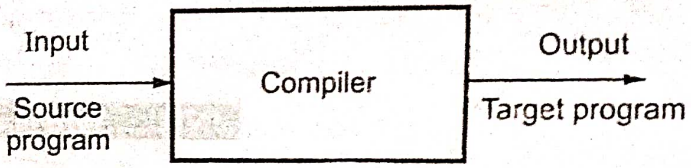


Fig. 1.2 Compiler

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

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.

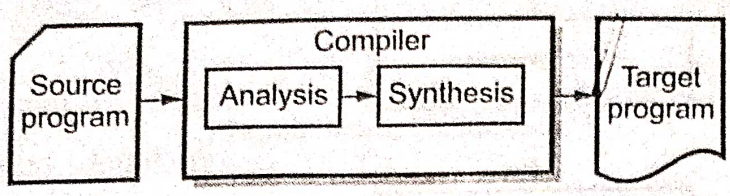


Fig. 1.3 Analysis and synthesis model

The analysis part is carried out in three sub parts

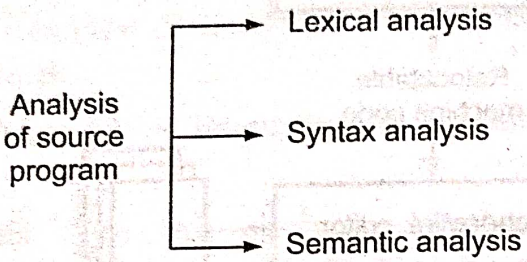


Fig. 1.4

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.

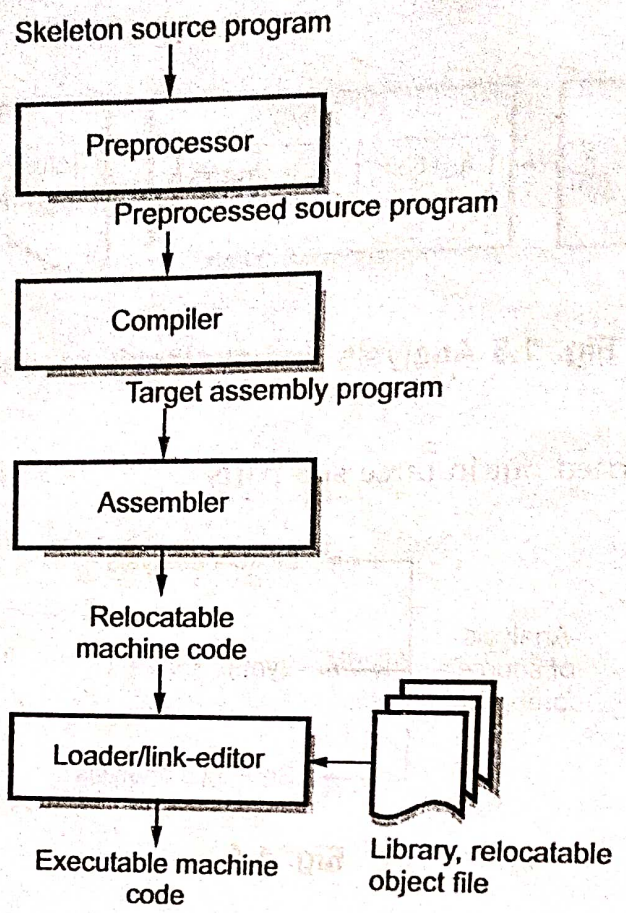


Fig. 1.5 (a) Process of execution of program

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

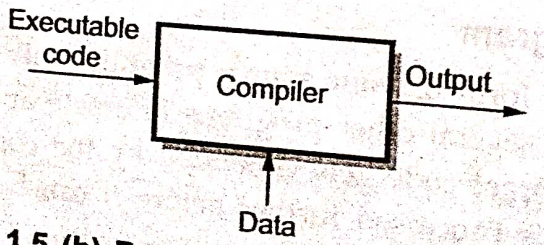


Fig. 1.5 (b) Process of execution of program

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

JNTU : Aug./Sep. 2007, 10 Marks

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 t complete source code is scanned and your source program is broken up into group strings called token. A token is a sequence of characters having a collective meaning. F 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 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 eliminat during the lexical analysis phase.

2. Syntax Analysis

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

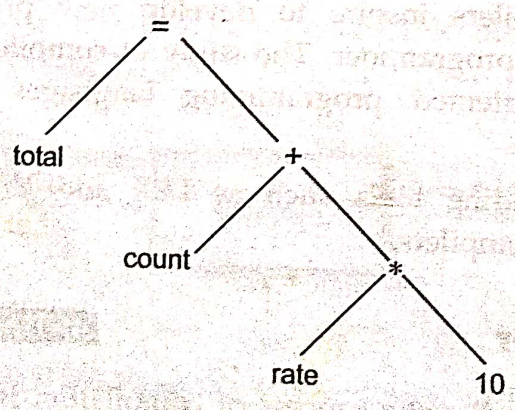


Fig. 1.6 Parse tree for total = count + rate *10

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

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,

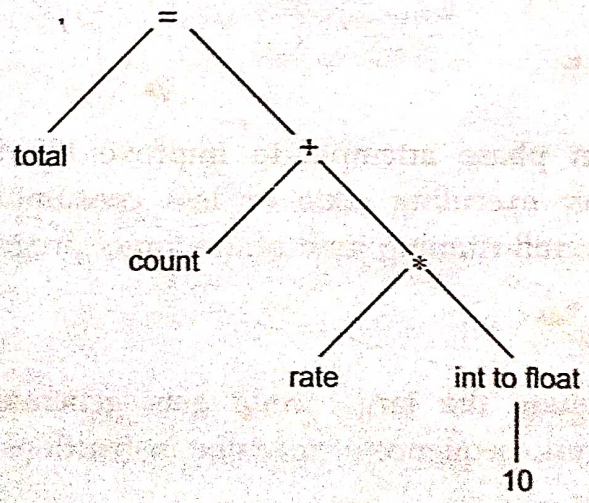


Fig. 1.7 Semantic analysis

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 be easily converted to target code. This code is in variety of forms such as *three address code, quadruple, triple, posix*. Here we will consider an intermediate code in three address code form. This is like an assembly language. The three address code consists of instructions 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 and those are,

1. Each three address instruction has at the most one operator in addition to the 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

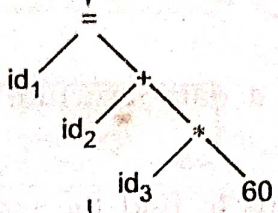
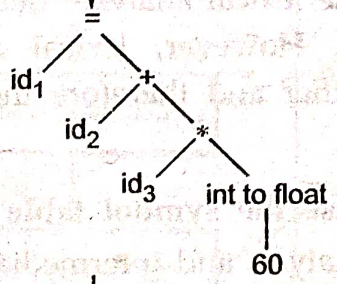
The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code or less consumption of memory. Thus by optimizing the code the overall running time of the target program can be improved.

6. Code Generation

In code generation phase the target code gets generated. The intermediate code 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.

	Input processing in compiler	Output
	a = b+c * 60	
	Lexical analyzer	
	$id_1 = id_2 + id_3 * 60$	Token stream
	Syntax analyzer	
		Syntax tree
	Semantic analyzer	
		Semantic tree
	Intermediate code generator	
	$t_1 := \text{inttofloat}(60)$ $t_2 := id_3 * t_1$ $t_3 := id_2 + t_2$ $id_1 := t_3$	Intermediate code
	Code optimizer	
	$t_1 := id_3 * 60.0$ $id_1 := id_2 + t_1$	Optimized code
	Code generator	

	MOVF id_3, R_2
	MULF #60.0, R_2
	MOVF id_2,R_1
	ADDF R_2, R_1
	MOVF R_1, id_1
		Machine code

a	----------
b	----------
c	----------
:	----------

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.

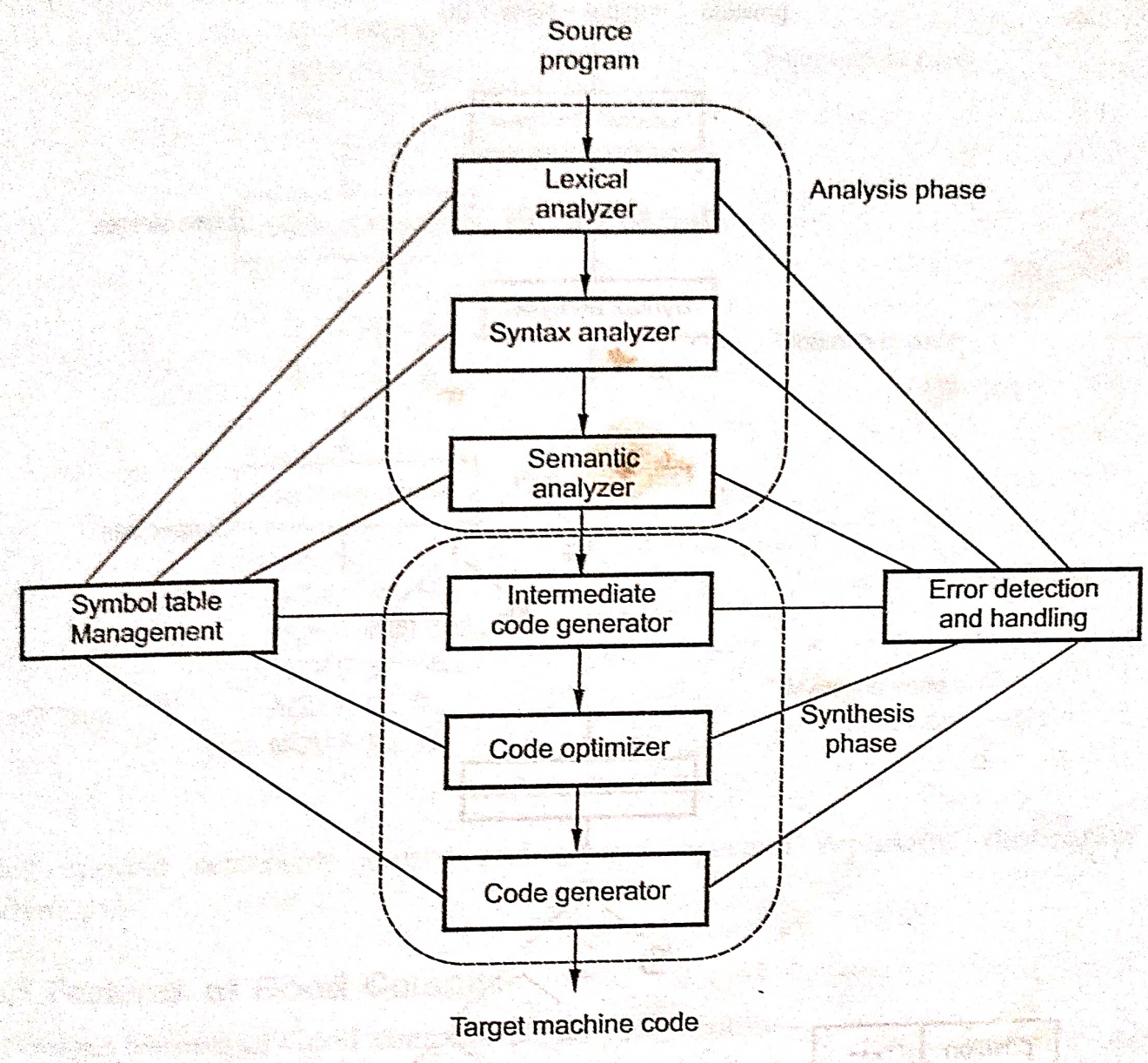


Fig. 1.8 Phases of compiler

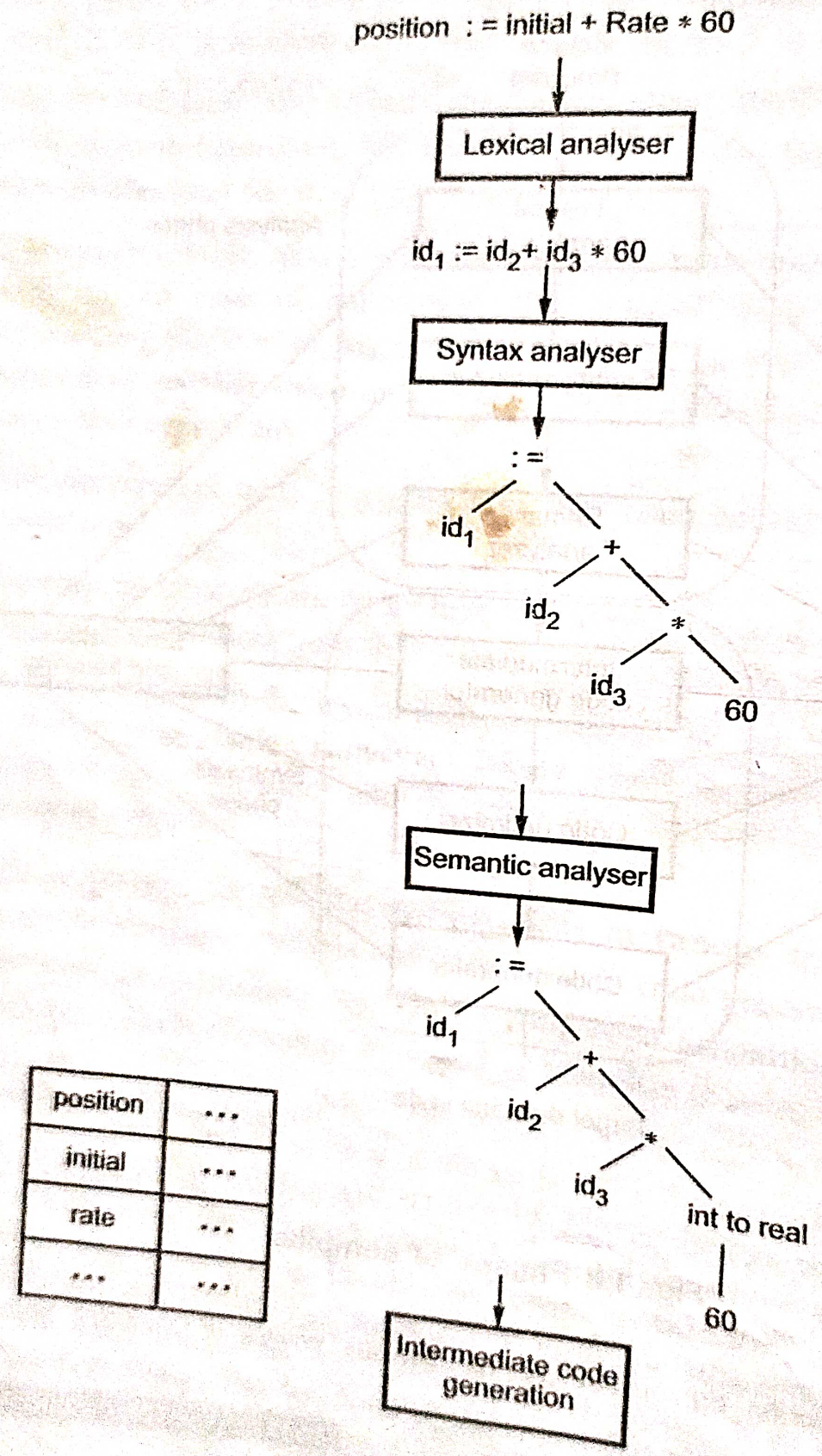
Example 1.1 : *Describe the output for the various phases of compiler with respect to following statements :*

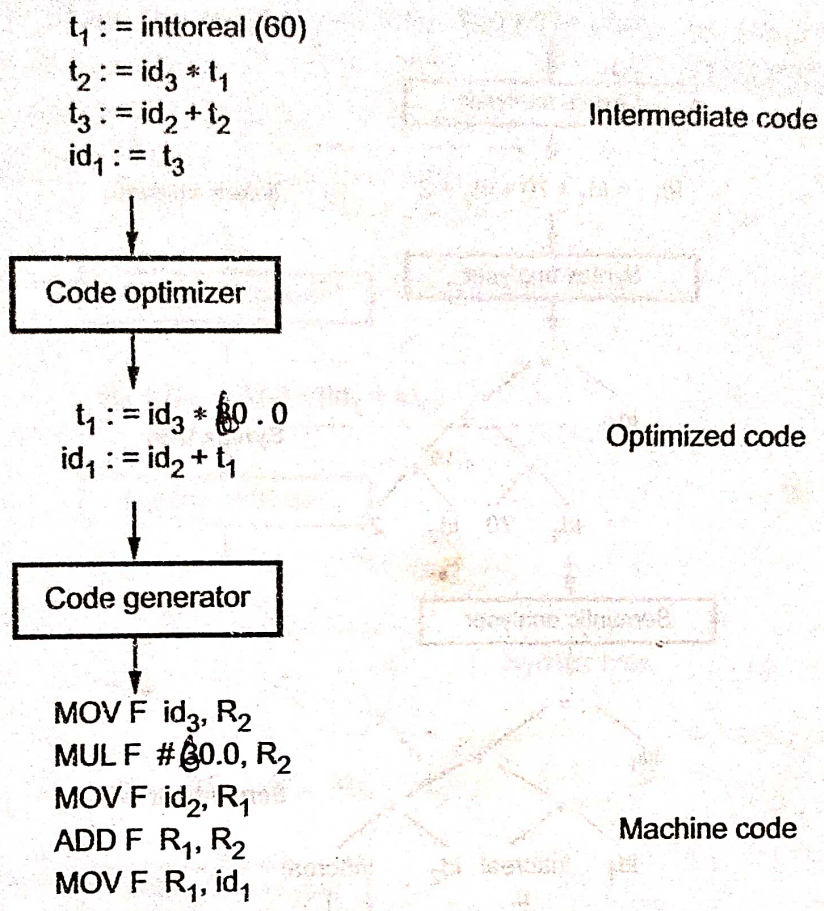
*position := initial + Rate * 60*

JNTU : April/May-2008, 10 Marks

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

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


position : = initial + Rate * 60
Lexical analyser
id1 := id2+ id3 * 60
Token stream
Syntax analyser
: =
id1
+
id2
*
id3
60
Syntax tree
Semantic analyser
: =
id1
+
id2
*
id3
int to real
60
Intermediate code generation
position ...
initial ...
rate ...
... ...



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

1.4.1 Features of Good Compiler

Various features of Good compiler are as given below -

1. The good compiler compiles the large amount code in **less time.**
2. The good compiler requires **less amount of memory** space to compile the source language.
3. The good compiler can **compile only the modified code** segment if frequent modifications are required in the source program.
4. While handling the hardware interrupts the good compilers **interact** closely with **operating system.**

Example 1.2 : *Consider the following fragment of 'C' code :*

```
float i, j ;
i = i * 70 + j + 2 ;
```

Write the output at all phases of the compiler for the above 'C' code.

JNTU : April/May-2008, January-2010, 10 Marks.

Solution :

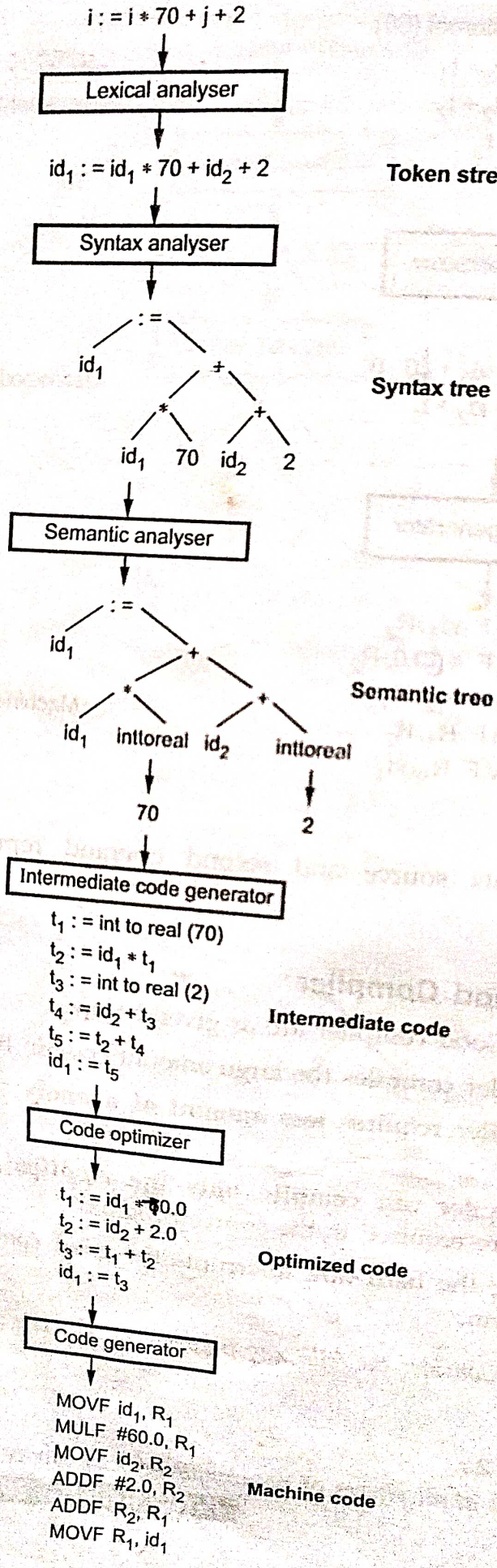


Fig. 1.9

1.5 Interpreter

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. 1.11.

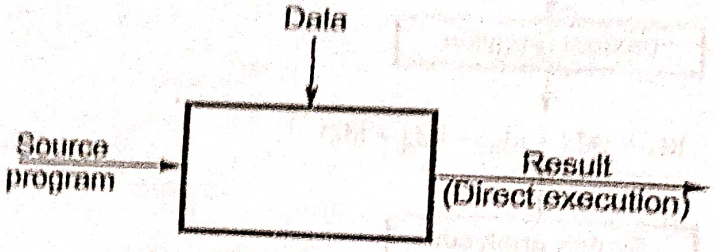


Fig. 1.11 Interpreter

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.

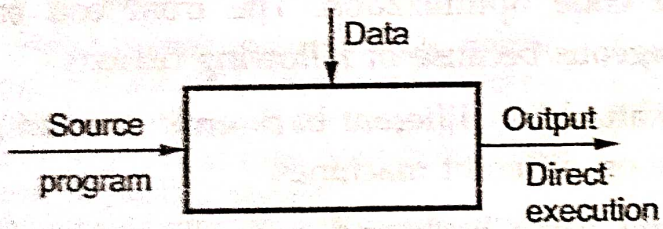
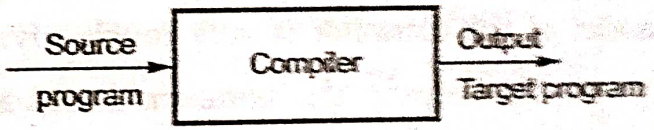
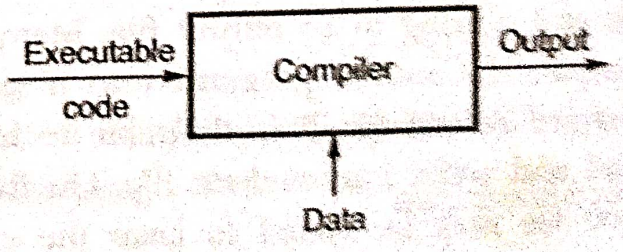
Disadvantages :

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

1.6 Comparison of Interpreter and Compiler

JNTU : April/May-2008, 6 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 on 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++.

1.7 Grouping of Phases

Front end back end

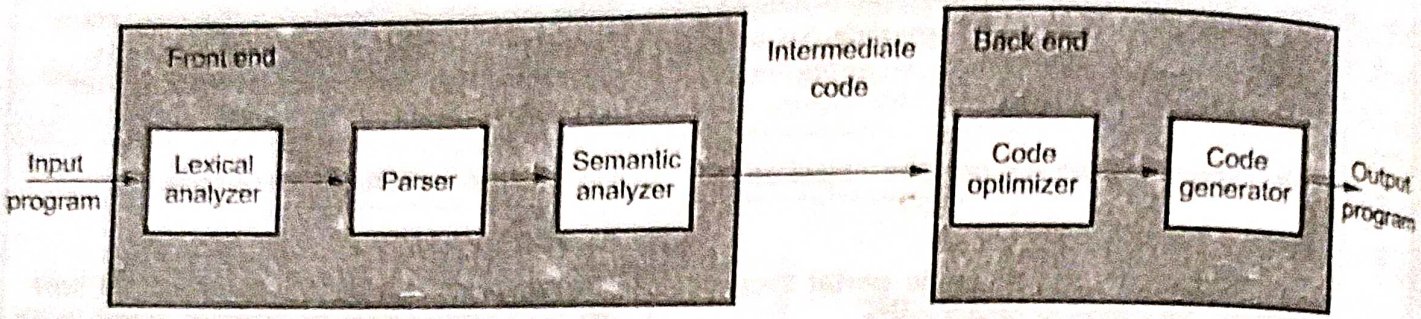


Fig. 1.15 Front end back end model

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 JNTU : April/May-2005, 2007, 6 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 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

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.

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

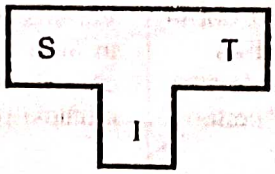


Fig. 1.13 (a) T diagram with S as source, T as target and I as implementation language

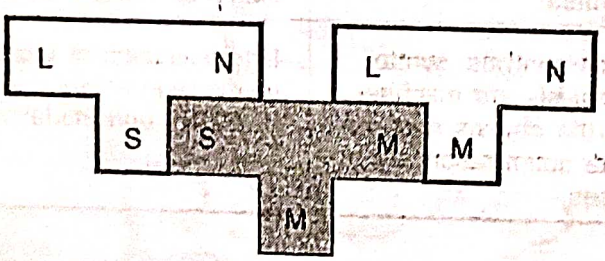


Fig. 1.13 (b) Cross compiler

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

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. 1.14.

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 -

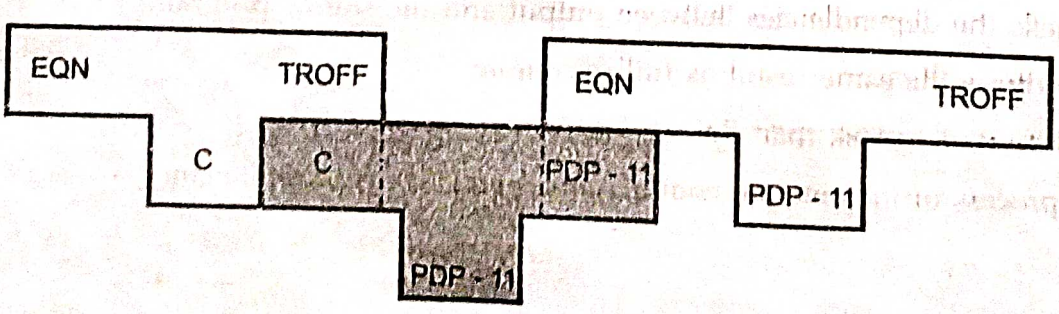


Fig. 1.14 Cross compiler for EQN

2.15 LEX

For efficient design of compiler, various tools have been built for constructing lexical analyzers using the special purpose notations called **regular expressions.**

The regular expressions are used in recognizing the tokens. Now we will discuss a special language that **specifies the tokens** using regular expressions. A tool called **LEX** gives this specification. Basically LEX is a unix utility which generates the lexical analyzer. A LEX lexer is very much faster in finding the tokens as compared to the handwritten LEX program in C.

LEX scans the source program in order to get the stream of tokens and these tokens are related together so that various programming constructs such as expressions, block statements, procedures, control structures can be realised. This task of relating the tokens together is known as **parsing**. During parsing of the program the rules are defined to establish the relationship between the tokens. These rules are called **grammar**. The YACC - Yet Another Compiler Compiler is another automated tool which is used to specify the grammar for realising the source programming construct. YACC takes the description of a grammar in some specification file and produces the C routine called parser. Thus LEX and YACC are the two important utilites that generate the Lexical analyzer and Syntax analyzers.

Let us first go for understanding of LEX.

2.16 The Simplest LEX Program

The LEX specification file can be created using the extension .l(often pronounced as dot L). For example, the specification file can be **x.l** . This **x.l** file is then given to LEX compiler to produce lex.yy.c. This lex.yy.c. is a C program which is actually a lexical analyzer program. The LEX specification file stores the regular expressions for the tokens and the lex.yy.c. file consists of the **tabular representation** of the transition diagrams constructed for the regular expression. The lexemes can be recognized with the help of this tabular representation of transition diagram.

In specification file **LEX actions** are associated with every regular expression. These actions are simply the pieces of C code. This C code is then directly carried over to the lex.yy.c. Finally the compiler compiles this generated lex.yy.c and produces an object program **a.out.** When some input stream is given to a.out then sequence of tokens get generated. The above described scenario can be modelled below.

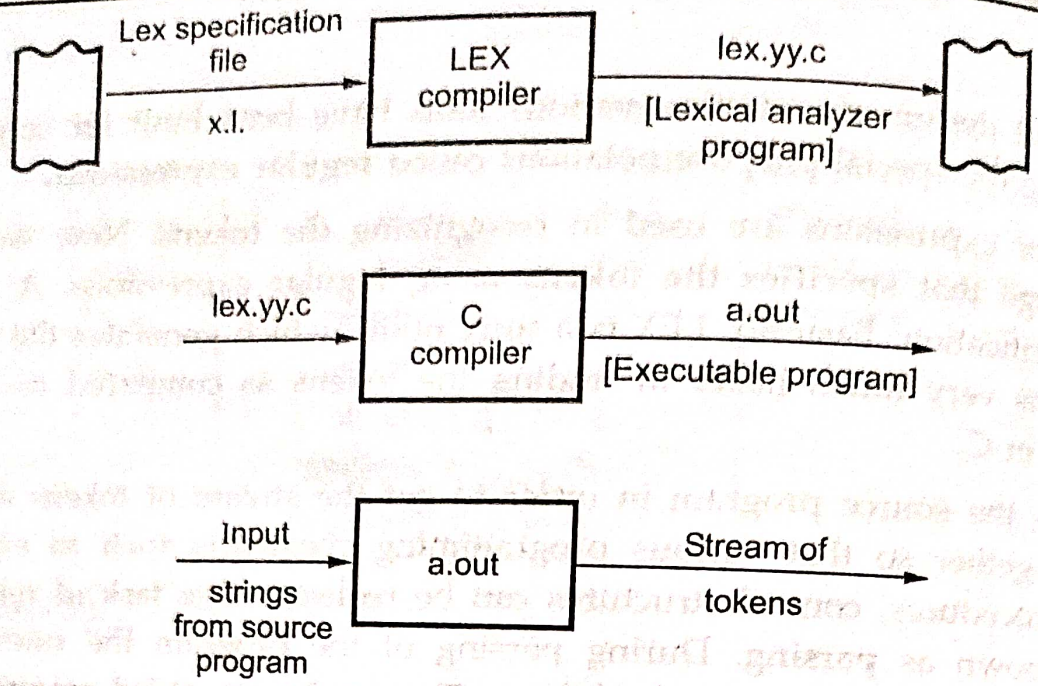


Fig. 2.35 Generation of lexical analyzer using LEX

2.16.1 Recognizing Words with LEX

Now the question arises how do we write the specification file? Well, the L program consists of three parts -

1. Declaration section
2. Rule section and
3. Procedure section.

```
%{
Declaration section
%}
%%
Rule section
%%
Auxiliary procedure section
```

- In the **declaration section** declaration of variable constants can be done. Some regular definitions can also be written in this section. The regular definitions are basically components of regular expressions appearing in the rule section.
- The **rule section** consists of regular expressions with associated actions. These translation rules can be given in the form as -

R_1 {action$_1$}
R_2 {action$_2$}
.
.
.
R_n {action$_n$}

Where each $\mathbf{R_i}$ is a regular expression and each **action$_i$** is a program fragment describing what action is to be taken for corresponding regular expression. These actions can be specified by piece of C code.

- And third section is a **auxiliary procedure** section in which all the required procedures are defined. Sometimes these procedures are required by the actions in the rule section.
- The lexical analyzer or scanner works in co-ordination with parser. When activated by the parser, the lexical analyzer begins reading its remaining input, character by character at a time. When the string is matched with one of the regular expressions R_i then corresponding action$_i$ will get executed and this action$_i$ returns the control to the parser.
- The repeated search for the lexeme can be made in order to return all the tokens in the source string. The lexical analyzer ignores white spaces and comments in this process. Let us learn some LEX programming with the help of some examples -

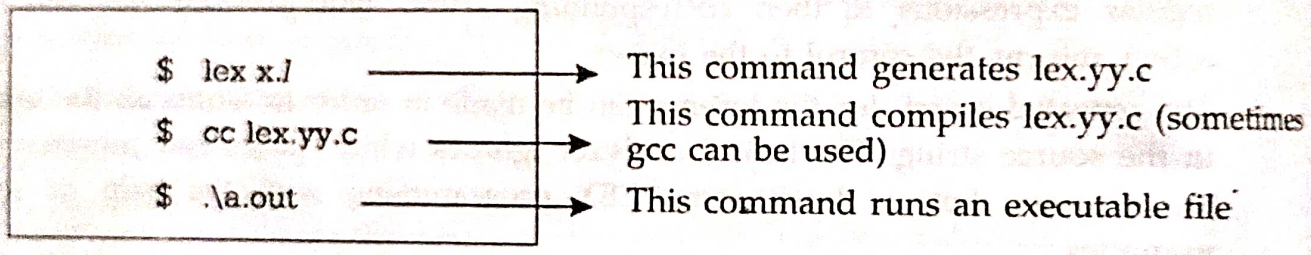
```
%{
%}
%%
"Rama" |
"Seeta" |
"Geeta" |
"Neeta" |    printf("\n Noun");
"Sings" |
"dances" |
"eats"       printf ("\n Verb");
%%
main ()
{
   yylex();
}
int yywrap()
{
   return 1;
}
```

This is a simple program that recognizes noun and verb from the string clearly. There are 3 sections in above program.

- The section starting and ending with %, { and %} respectively is a **definition section**.
- The section starting with %% is called **rule section**. This section is closed by %%. Within %% consists of regular expressions and actions. Rule 1 gives the definition of noun and second rule gives the definition of verb.
- The third section consists of two functions the **main function** and the **yywrap function**. In main function call to **yylex** routine is given. This function is defined in lex.yy.c program. First we will compile our above program (x.*l*) using lex compiler and then the lex compiler will generate a ready C program named lex.yy.c. This lex.yy.c makes use of regular expression and corresponding actions defined in x.*l*. Hence our above program x.*l* is called lex specification file.

When we compile lex.yy.c file using command cc, here cc means compile C. We get an output file named a.out. (This is a default output file on LINUX platform). On execution of a.out we can give the input string.

Following commands are used to run the lex program x.*l*.



After entering these commands a blank space for entering input gets available. Then we can give some valid input.

```
Rama eats
Noun
verb
Seeta sings
Noun
verb
```

Then press either **control + c** or **control + d** to come out of the output.

vi Editor commands

While writing LEX program on LINUX popularly a **vi editor** is used. Here are some commonly used vi commands.

- To start vi editor : At the command prompt we can give following command
  $ vi filename

For example, $ vi x.*l*.

2.23 Classification of Errors

The errors can be classified as

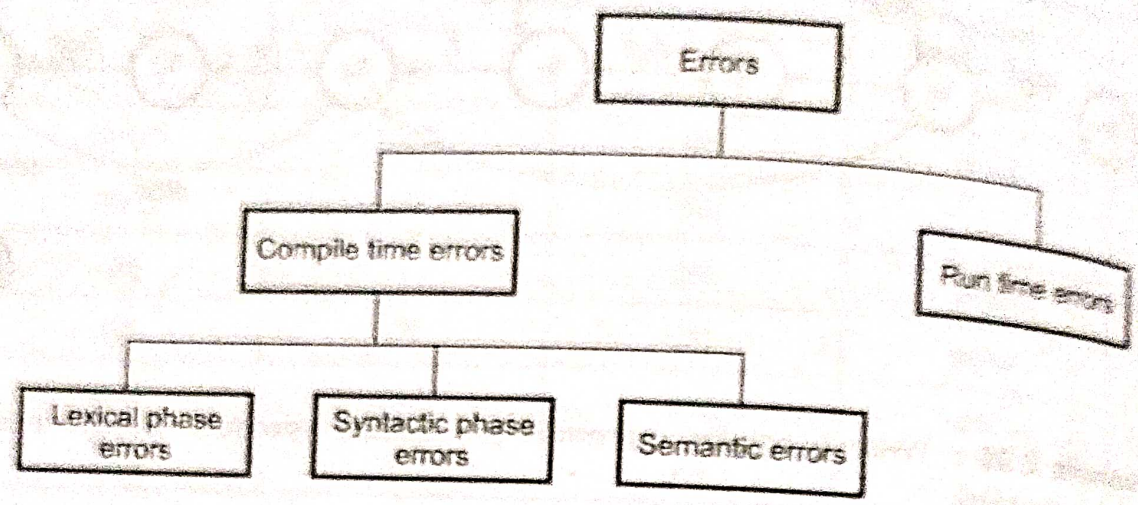


Fig. 2.38 Classification of errors

Globally there are two types of errors : Compile time and run time errors.

During the phases of compiler these errors can be potentially detected hence they are classified by names of compilation phases.

2.22.1 Lexical Phase Errors

These type of errors can be detected during lexical analysis phase. Typical lexical phase errors are

- Spelling errors. Hence get incorrect tokens.
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters.

Normally due to typing mistakes the wrong spellings may appear in the program. This causes a lexical error.

For example

```
swtich (choice)
{
.....
.....
}
```

In above code 'switch' can not be recognized as a misspelling of keyword switch. Rather lexical analyzer will understand that it is an identifier and will return it as valid identifier. Thus misspelling causes errors in token formation. This is can be a possibility of an error which occurs due to cascading of characters.

For example

```
switch (choice)
{
case1:   get_data();
         break;
case2:   display();
         break;
}
```

In above given 'C' code case 1 will not be identified because there is no space between 'case' and '1'. But the compiler does not report any error message as it identifies 'case1' as valid identifier. This type of error is lexical phase error.

Consider

```
printf("\n Hello India");$
```

This is also a lexical error as an illegal character '$' appears at the end of the statement.

If length of identifiers get exceeded the error occurs. For example if in FORTRAN there is an identifier whose length is of 10 characters then lexical error occurrs.

INPUT BUFFERING:

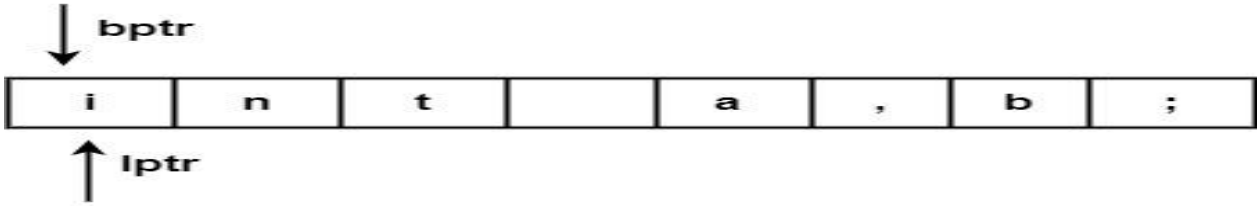
Lexical Analysis has to access secondary memory each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis.

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

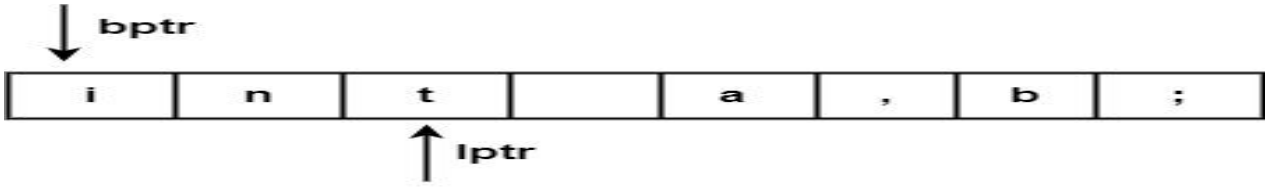
- **Begin Pointer (bptr)** − It points to the beginning of the string to be read.
- **Look Ahead Pointer (lptr)** − It moves ahead to search for the end of the token.

Example − For statement int a, b;

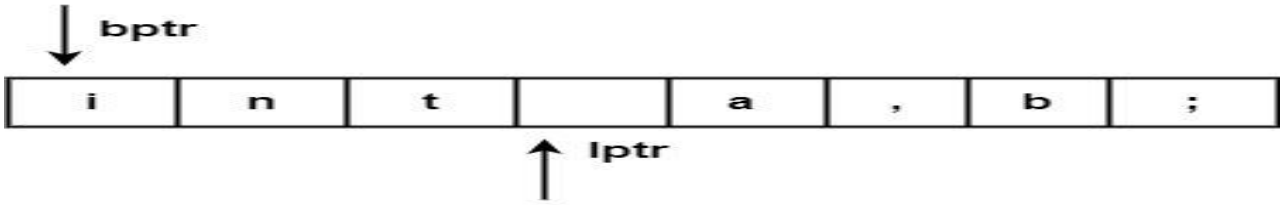
- Both pointers start at the beginning of the string, which is stored in the buffer.



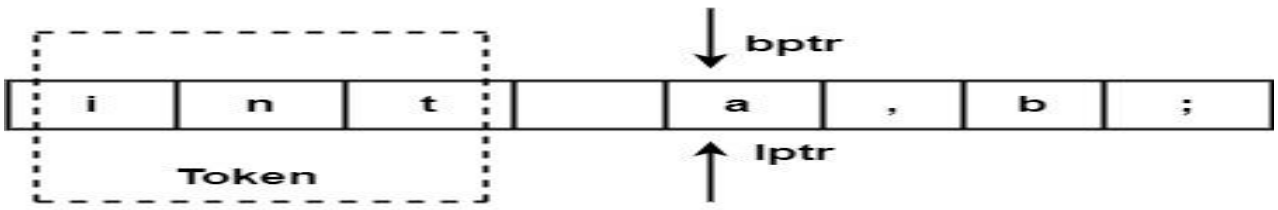
- Look Ahead Pointer scans buffer until the token is found.



- The character ("blank space") beyond the token ("int") have to be examined before the token ("int") will be determined.



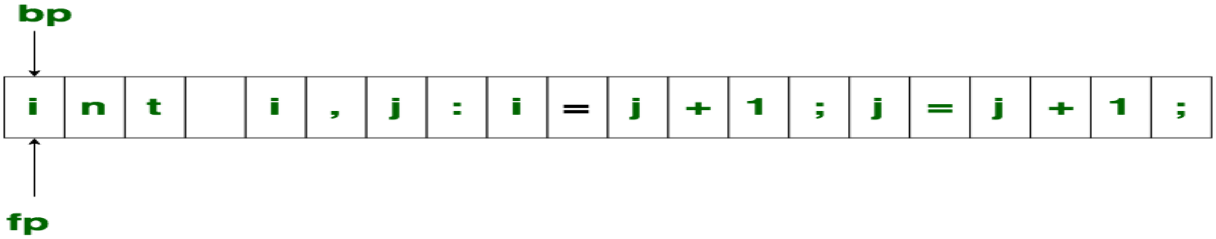
- After processing token ("int") both pointers will set to the next token ('a'), & this process will be repeated for the whole program.



A buffer can be divided into two halves. If the look Ahead pointer moves towards halfway in First Half, the second half is filled with new characters to be read. If the look Ahead pointer moves towards the right end of the buffer of the second half, the first half will be filled with new characters, and it goes on.

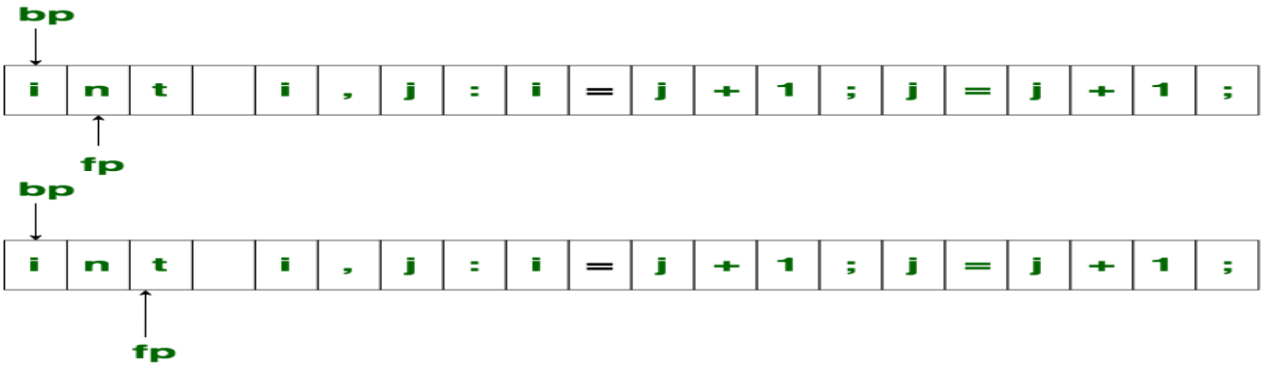
Input Buffering

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward to keep track of the pointer of the input scanned.



Initial Configuration

Initially both the pointers point to the first character of the input string as shown below



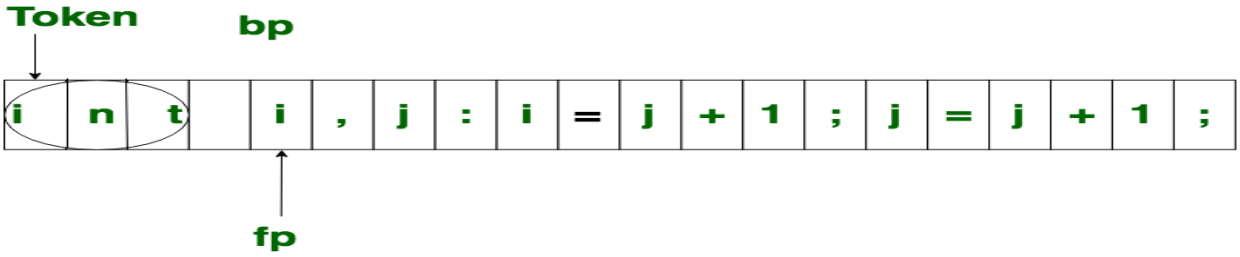
Input Buffering

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified.

The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.

The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A

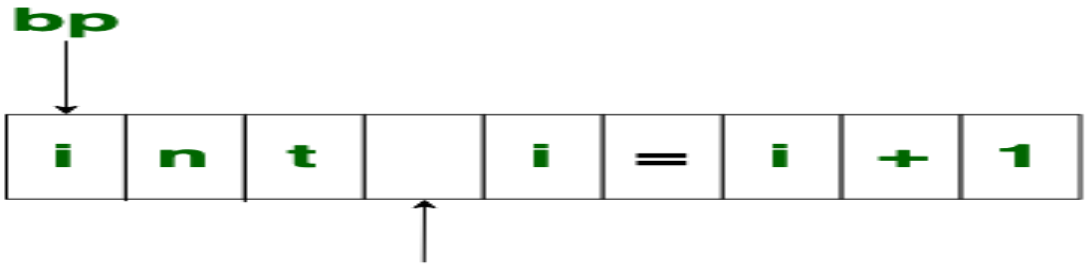
block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Input buffering

1. **One Buffer Scheme:**

   In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



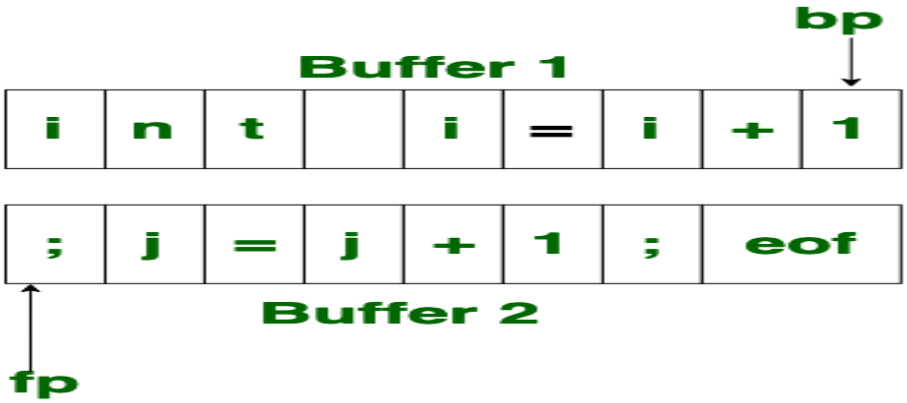
One buffer scheme storing input string

2. **Two Buffer Scheme:**

   To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely.

   Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer.

Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.



Two buffer scheme storing input string