UNIT – 1

INTRODUCTION TO COMPILING

1.1 INTRODUCTION

TRANSLATORS:

A translator is one kind of program that takes one form of program as the input and converts it into another form. The input program is called source language and the output program is called target language.

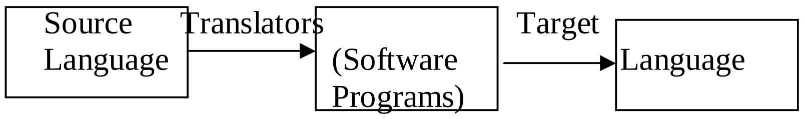


Fig 1.1: Translators

Some example translators are Assemblers, Compilers and Interpreters

ASSEMBLERS:

It is a translator for an assembly language of a computer. It is the lowest level programming language. So it is machine dependent.

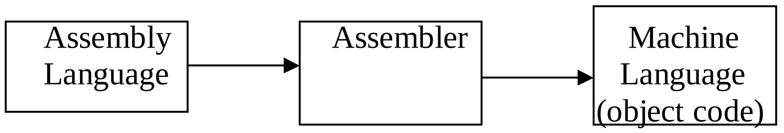


Fig 1.2: Assembler

COMPILERS:

A compiler is a program that reads a program in one language, the source language and translates into an equivalent program in another language, the target language. The translation process should also report the presence of errors in the source program.

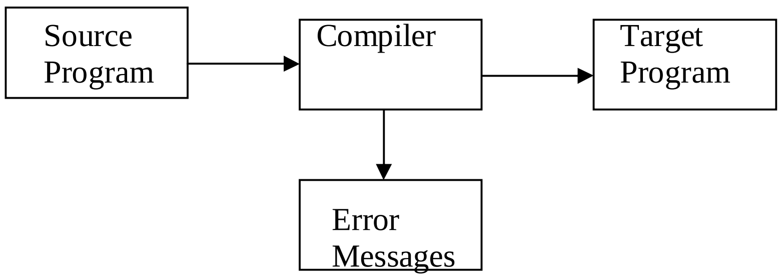


Fig 1.3: Compiler

INTERPRETER:

It reads a source program written in high level programming language line by line basis rather than producing object code. It needs more memory during execution. Some of the difference between compilers and interpreters are listed below.

INTERPRETER	COMPILER
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.
The interpreter s do not produce object code.	The compilers produce object code.
Merit: The interpreters can be made portal because they do not produce object code.	**Demerits:** The compiler has to present on the host machine when particular needs to be compiled.
Merits: Interpreters are simpler and give us improved debugging environment.	**Demerits:** The compiler is a complex program and it requires large amount of memory.

Table1.1: Difference between interpreter and compiler

LOADER:

Loader loads the binary code in the memory ready for execution. It transfers the control to the first instruction. It is responsible for locating program in main memory every time it is being executed.

There are various loading schemes.

1. **Assemble and go loader:** The assembler simply places the code into memory and the loader executes a single instruction that transfers control to the starting instruction of the assembled program.
2. **Absolute loader:** Object code must be loaded into absolute addresses in the memory to run.
3. **Relocating loader:** This loader modifies the actual instruction of the program during the process of loading a program.

LINKER:

It is the process of combining various pieces of code and data together to form a single executable file.

Two types of linking are

Static Linking: All references are resolved during loading at linkage time.

Dynamic Linking: References are resolved during run time. It take advantage of the full capabilities of virtual memory.

Loader is a program which performs two functions: loading and link editing. Loading is a process in which the re-locatable machine code is read and the re-locatable addresses are altered. Then that code with altered instructions and data is placed the memory at proper location. The job of link editor is to make a single program from several files of re-locatable machine code. If code in on file refers the location in another file then such a reference is called external reference.

SOFTWARE TOOLS :

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

1) **Structure editor:**
- ✓ Takes as input a sequence of commands to build a source program.
- ✓ Structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- ✓ For example, it can supply key words automatically - while …. do and begin….. end.

2) Pretty printers :
- ✓ A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
- ✓ For example, comments may appear in a special font.

3) Static checkers :
- ✓ A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
- ✓ For example, a static checker may detect that parts of the source program can never be executed.

4) Interpreters :
- ✓ Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
- ✓ An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
- ✓ Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.

The following are similar to conventional compilers

Text formatters.

It takes input as a stream of characters, most of which is text to be type set, but some of which includes commands to indicate paragraphs, figures or mathematical structures like subscripts and superscripts.

Silicon Compiler.

It has a source language that is similar or identical to a conventional programming language. The variable of the language represent logical signals (0 or 1) or group of signals in a switching circuit. The output is a circuit design in an appropriate language.

Query interpreters.

It translates a predicate containing relational and Boolean operators into commands to search a database or records satisfying that predicate.

1.2 ANALYSIS OF THE SOURCE PROGRAM

The source program is analyzed by the compiler to check whether the program is up to the standard of the corresponding programming language or not. A compiler is composed of several components called phases, each performing one specific task. A complete compilation procedure can be divided into six phases and these phases can be regrouped into two parts.

They are,

1. **Analysis part**
2. **Synthesis Part**

The **analysis part** breaks up the source program into constant piece and creates an intermediate representation of the source program. Analysis can be done in three phases.

- ✓ Lexical Analysis
- ✓ Syntax Analysis (or) Hierarchical analysis
- ✓ Semantic Analysis

The **synthesis part** constructs the desired target program from the intermediate representation. Synthesis consists of three phases:

- ✓ Intermediate code generation
- ✓ Code Optimization
- ✓ Code generator.

Linear analysis (Lexical analysis or Scanning):

The lexical analysis phase reads the characters in the source program and grouped into them as tokens. Tokens are sequence of characters having a collective meaning.

Example:position: = initial + rate * 60.
This statement is grouped into 7 tokens.
Identifiers: position, initial, rate.
Assignment symbol: =
Operators: +, *
Constant: 60
Blanks: eliminated.

Hierarchical analysis (Syntax analysis or Parsing):

It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output.

Example : position : = initial + rate * 60

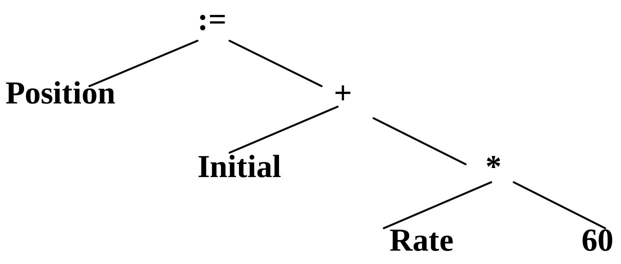


Fig 1.4 Syntax Tree (or) Parse Tree

Semantic analysis:
In this phase checks the source program for semantic errors and gathers type information for subsequent code generation phase. An important component of semantic analysis is type checking.

Example : int to real conversion.

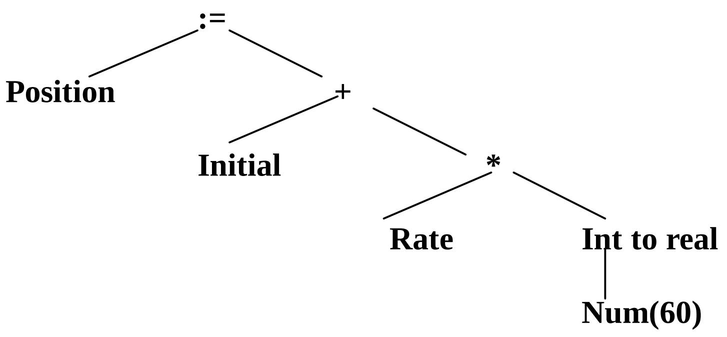


Fig 1.5.Semantic tree

1.2.1 PHASES OF COMPILER

The compiler has a number of phases plus symbol table manager and an error handler. The first three phases, forms the bulk of the analysis portion of a compiler. Symbol table management and error handling, are shown interacting with the six phases.

Phase: A phase is a logically organized operation that takes input as one representation of source program and produces output as another representation without changing its meaning.

Pass: Portion of one or more phases are combined into a module called PASS. A pass reads the source program of output of previous pass and makes the transformations of current pass and finally writes the output of the current pass into an intermediate file. This intermediate file can be used by the subsequent passes.

Symbol table management (or) Book keeping:
An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.

For example: The symbol table entries for the statement **Position=initial + rate * 60** are

Symbol table		
Lexeme	**Token**	**Attribute**
Position	Identifier	Pointer

=	Operator	-
Initial	Identifier	Pointer
+	Operator	-
Rate	Identifier	Pointer
*	Operator	-
60	Constant	60

Table 1.2: Symbol Table

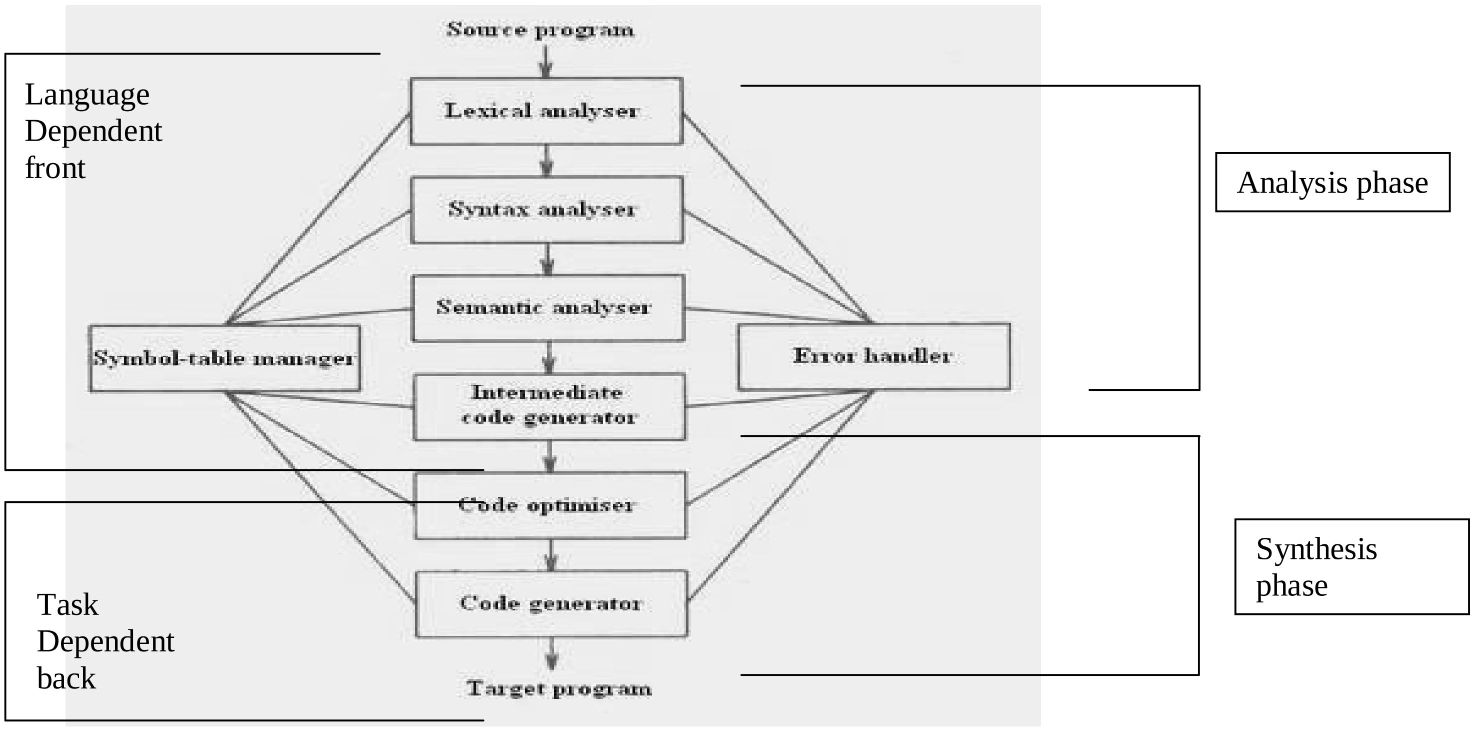


Fig 1.6: Phases of Compiler (or) Logical structure of a compiler

The Analysis phases:

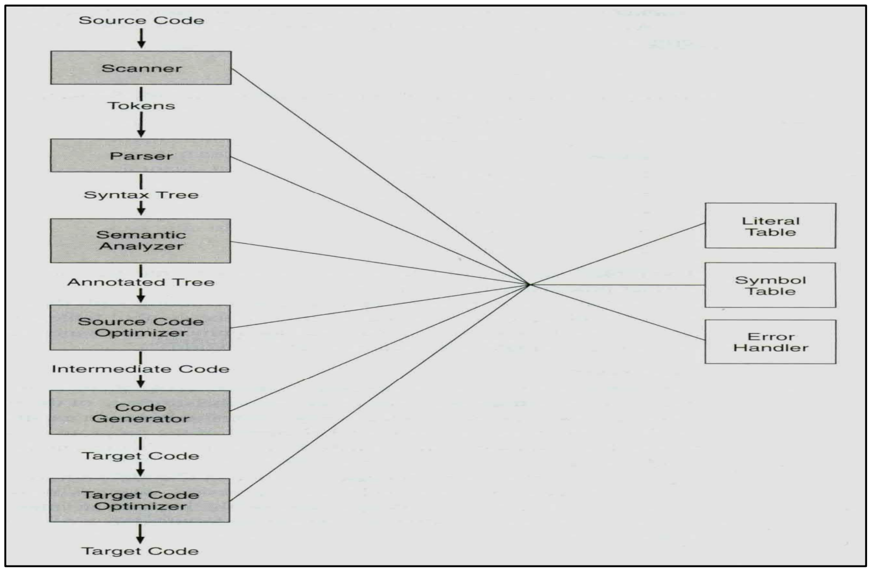
As translation progresses, the compiler's internal representation of the source program changes. Consider the statement,

position := initial + rate * 10

The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword etc.

The character sequence forming a token is called the ***lexeme***for the token. Certain tokens will be augmented by a 'lexical value'. For example, for any identifier the lexical analyzer generates not only the token identifier but also enters the lexeme into the symbol table, if it is not already present there. The lexical points to the symbol table entry for this lexeme. The representation of the statement given above after the lexical analysis would be:

id1: = id2 + id3 * 10



Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax Trees.

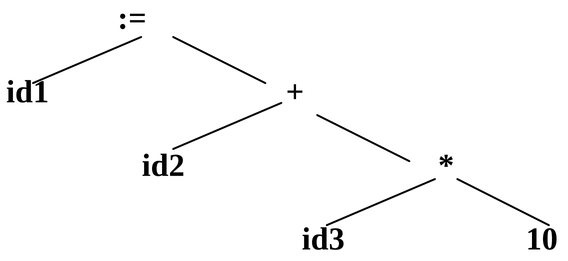


Fig 1.7: Syntax tree

Intermediate Code Generation:

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can have a variety of forms. In three-address code, the source program might look like this,

```
temp1: = inttoreal (10)
temp2: = id3 * temp1
temp3: = id2 + temp2
id1: = temp3
```

Code Optimization:

The code optimization phase attempts to improve the intermediate code, so that faster running machine codes will result. Some optimizations are trivial. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called **'optimizing compilers',** a significant fraction of the time of the compiler is spent on this phase.

Two types of optimizations are being carried out by this phase.

- **Local Optimization**

Elimination of common sub expressions by using copy propagation.

- ❖ **Loop Optimization**
  Finding out loop invariants and avoiding them.

Code Generation:

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

Example:

```
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1
```

Error Detection and Reporting:

Each phase can encounter errors. Detecting and reporting of errors in source program is main function of compiler.

The various errors occur at different levels may be..

- ❖ The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.
- ❖ Errors when the token stream violates the syntax of the language are determined by the syntax analysis phase.
- ❖ During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.
- ❖ The intermediate code generator may detect an operator whose operands have incompatible types.
- ❖ The code optimizer, doing control flow analysis may detect that certain statements can never be reached.
- ❖ The code generator may find a compiler created constant that is too large to fit in a word of the target machine.
- ❖ While entering information into the symbol table, the book keeping routine may discover an identifier that has multiple declarations with contradictory attributes.

For Example:

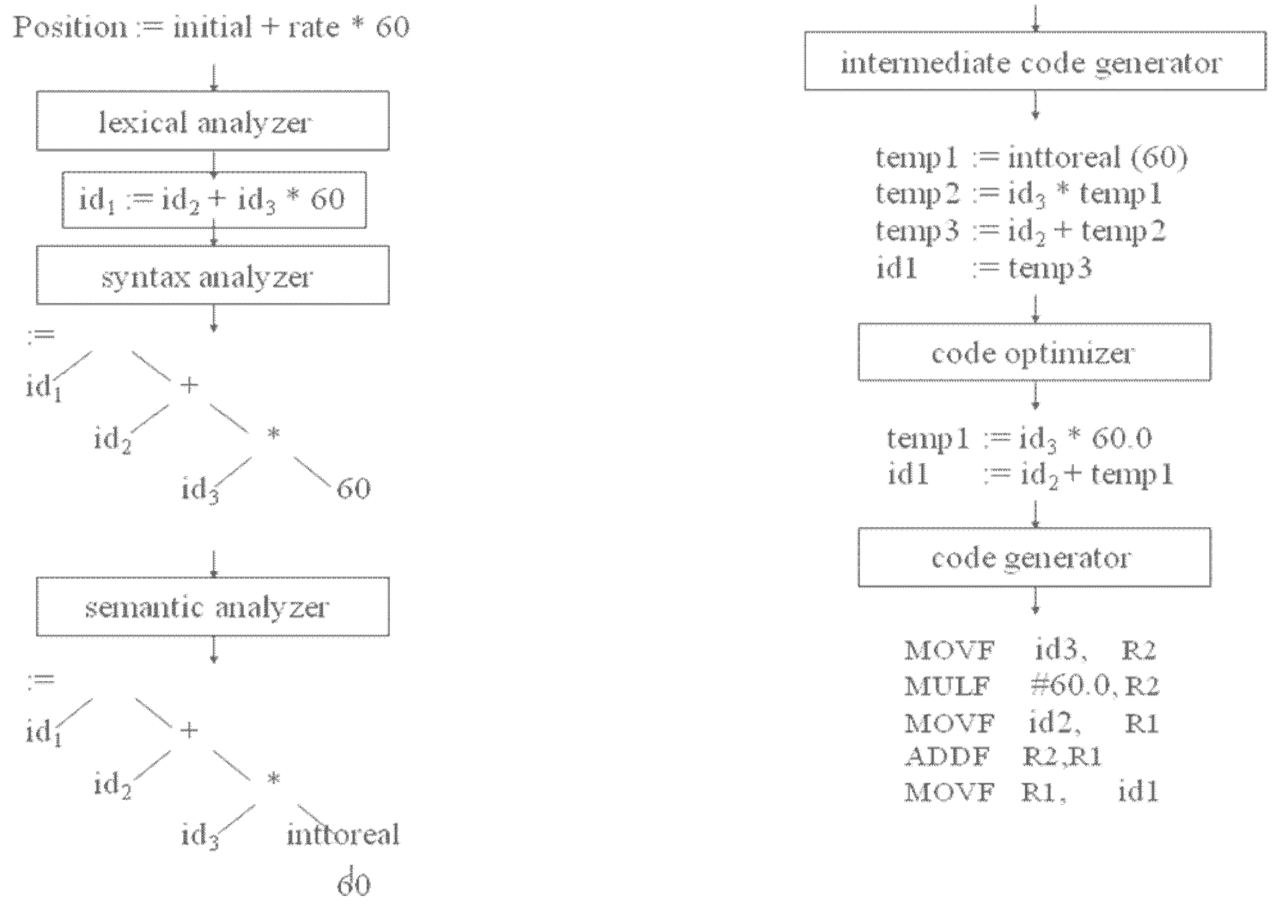
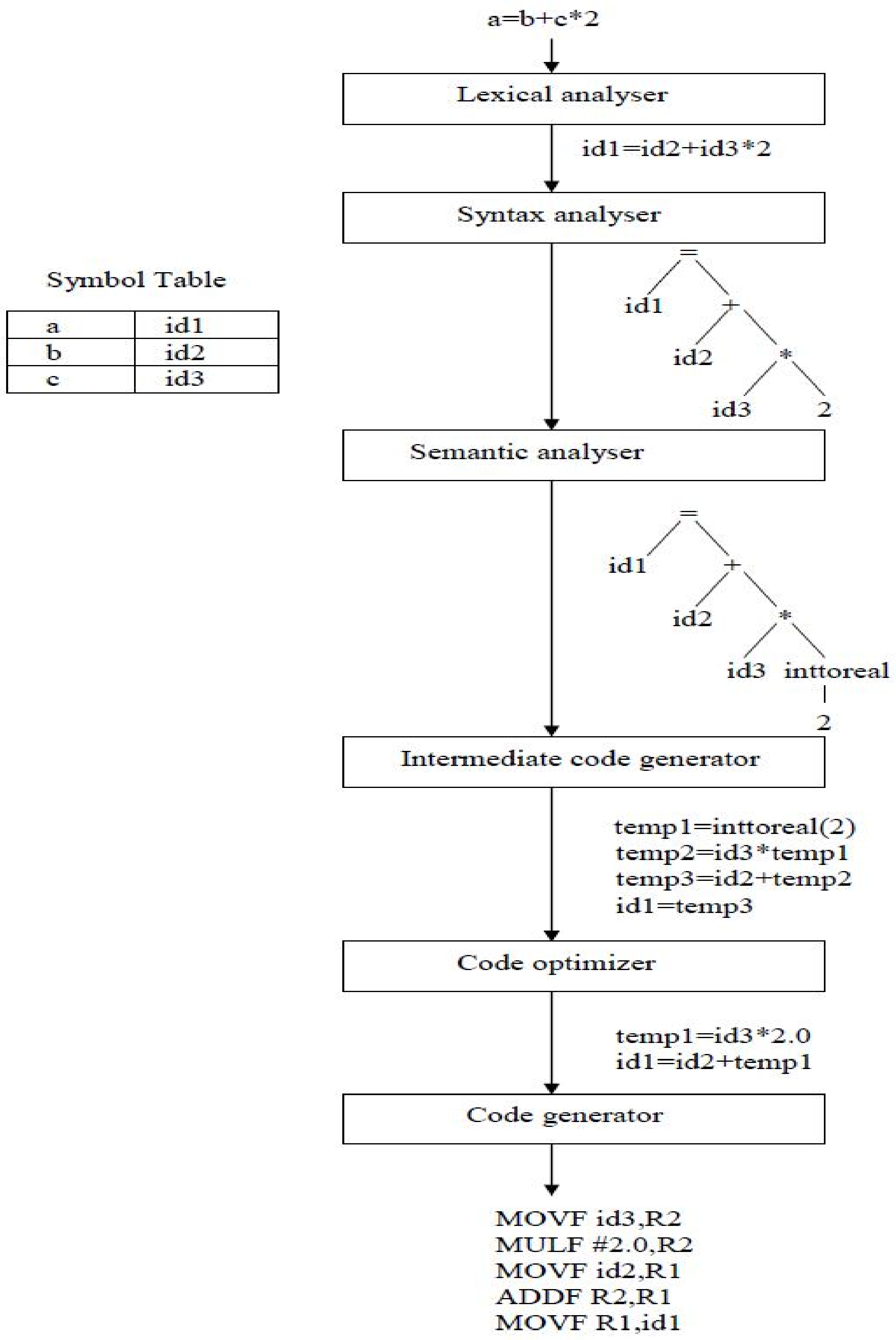


Fig 1.11: Translation of the statement **position = initial + rate * 60**

1.Find the output of each phase for the statement a= b+c*2



a	id1
b	id2
c	id3

```
temp1=inttoreal(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
```

```
temp1=id3*2.0
id1=id2+temp1
```

```
MOVF id3,R2
MULF #2.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1
```

COUSINS OF COMPILER.

The cousins of the compiler are

1. **Preprocessor.**
2. **Assembler.**
3. **Loader and Link-editor.**

PREPROCESSOR

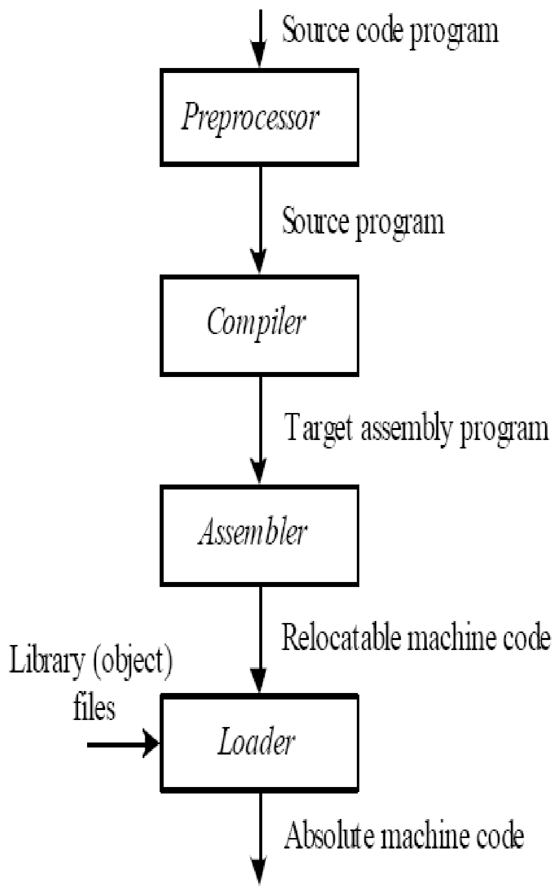


Fig 1.8: A language preprocessing system

A **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers. The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

They may perform the following functions

1. **Macro processing**
2. **File Inclusion**
3. **Rational Preprocessors**
4. **Language extension**

1. MACRO PROCESSING:
A **macro** is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure. The mapping processes that instantiates (transforms) a macro into a specific output sequence is known as ***macro expansion***.

macro definitions (#define, #undef)
To define preprocessor macros we can use #define. Its format is:
#define identifier replacement
When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. #define TABLE_SIZE 100
int table1[TABLE_SIZE];

int table2[TABLE_SIZE];
After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:
int table1[100];
int table2[100];

2.FILE INCLUSION:
Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

#include "file"
#include <file>
The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.

If the file name is enclosed between angle-brackets <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.
3. RATIONAL PREPROCESSORS:
These processors augment older languages with more modern flow of control and data structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.
4 .LANGUAGE EXTENSION:
These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language equal is a database query language embedded in C. Statements begging with ## are taken by the preprocessor to be database access statements unrelated to C and are translated into procedure calls on routines that perform the database access.
The behavior of the compiler with respect to extensions is declared with the #extension directive: #extension extension_name : behavior
#extension all : behavior
extension_name is the name of an extension. The token all means that the specified behavior should apply to all extensions supported by the compiler.
<u>ASSEMBLER</u>
Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into op-codes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution-e.g., to generate common short

sequences of instructions as inline, instead of *called* subroutines, or even generate entire programs or program suites.
There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- **One-pass assemblers** go through the source code once and assume that all symbols will be defined before any instruction that references them.
- **Two-pass assemblers** create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code. The assembler must at least be able to determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.

The advantage of a one-pass assembler is speed, which is not as important as it once was with advances in computer speed and capabilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in the program source. As a result, the program can be defined in a more logical and meaningful way. This makes two-pass assembler programs easier to read and maintain.

LINKERS AND LOADERS

A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks
1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references.
3. Resolves references among files.

Loader
A **loader** is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program. Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded programcode.All operating systems that support program loading have loaders, apart from systems where code executes directly from ROM or in the case of highly specialized computer systems that only have a fixed set of specialized programs.

In many operating systems the loader is permanently resident in memories, although some operating systems that support virtual memory may allow the loader to be located in a region of memory that is page-able.

In the case of operating systems that support virtual memory, the loader may not actually copy the contents of executable files into memory, but rather may simply declare to the virtual memory subsystem that there is a mapping between a region of memory allocated to contain the running program's code and the contents of the associated executable file. The virtual memory subsystem is then made aware that pages with that region of memory need to be filled on demand if and when program execution actually hits those areas of unfilled memory. This may mean parts of a program's code are not actually copied into memory until they are actually used, and unused code may never be loaded into memory at all.

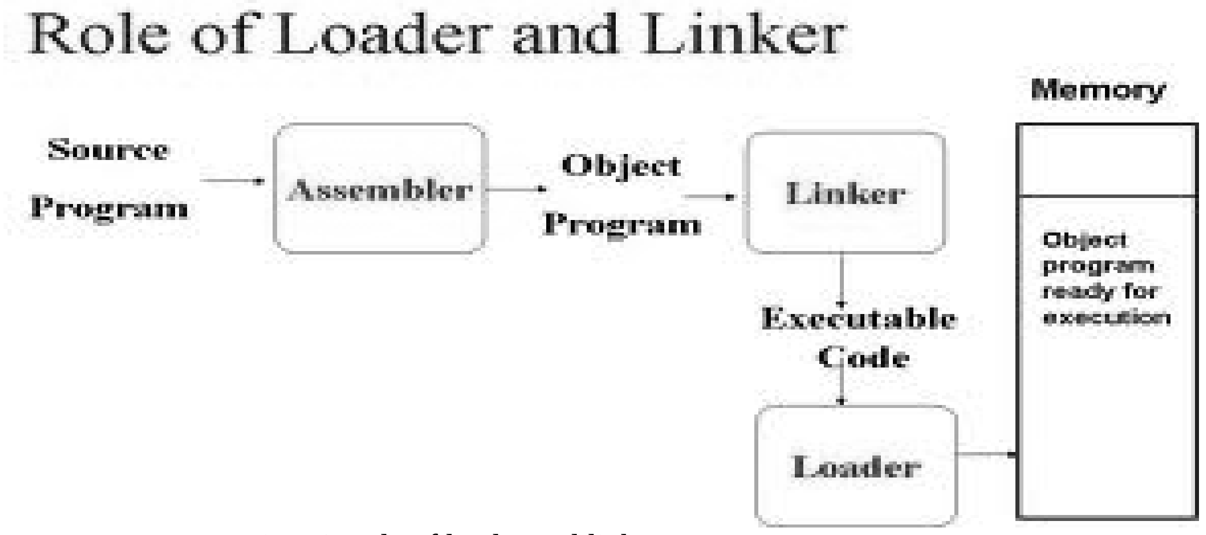


Fig 1.9 Role of loader and linker

GROUPING OF PHASES:

The structure of compiler can be viewed from many different angles. The structures of compiler have a major impact on its reliability, efficiency, usefulness and maintainability.

Front end and Back end:

Front end normally includes lexical, syntactic analysis, the creation of symbol table, semantic analysis and the generation of intermediate code. A certain amount of code optimization can be done by front end as well. Activities from several phases may be grouped together into **a pass** that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass).

Single pass: usually requires everything to be defined before being used in source program.

Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

COMPILER CONSTRUCTION TOOLS:

Some of the available built in tools for designing the phases of compiler are:

1. **Scanner generator:** These software automatically generate lexical analyzers from a specification of token formats based on regular expressions. The basic organization of the lexical analyzer produced by this software is a finite automation(for scanner).It consumes a large fraction of the running time of a compiler.
   Example-YACC (Yet Another Compiler-Compiler).

2. **Parser Generator:** These software produce syntax analyzers from input that represents the syntax of programming language based on context free grammar(for Parser).

3. **Syntax directed translation engines**: These produce collections of software that analyses the parse tree generating intermediate code(for intermediate code generation). Each translation is defined in terms of translations at its neighbor nodes in the tree.

4. **Data flow Engines:** These produce code optimizer for efficient use of resources and reduces the execution time by gathering information about the way of transmitting values from one part of a program to another part(for code optimization).

5. **Automatic code Generator:** These produce code Generator for generating target code from the intermediate representation. It takes into account the collection of rules that define the translation of each type intermediate state into target language.

LEXICAL ANALYZER :

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis. As in the figure, upon receiving a “get next token” command from the parser the lexical analyzer reads input characters until it can identify the next token.

The Role of the Lexical Analyzer

- Read input characters
- To group them into lexemes
- Produce as output a sequence of tokens
  - input for the syntactical analyzer
- Interact with the symbol table
  - Insert identifiers
- to strip out
  - comments
  - whitespaces: blank, newline, tab, …
  - other separators
- to correlate error messages generated by the compiler with the source program
  - to keep track of the number of newlines seen
  - to associate a line number with each error message

Issues in Lexical Analysis:

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

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

2) Compiler efficiency is improved.

3) Compiler portability is enhanced.

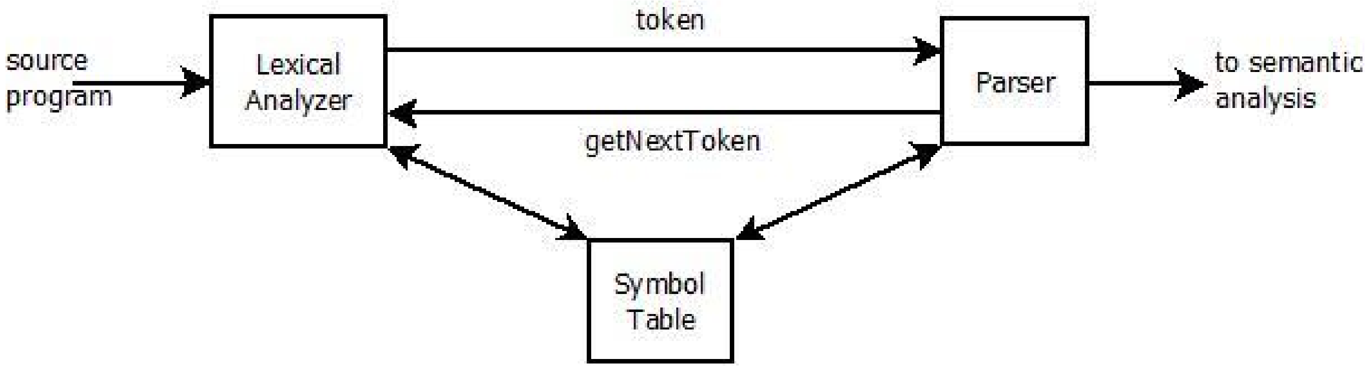


Fig 1.10: Role of Lexical Analyzer

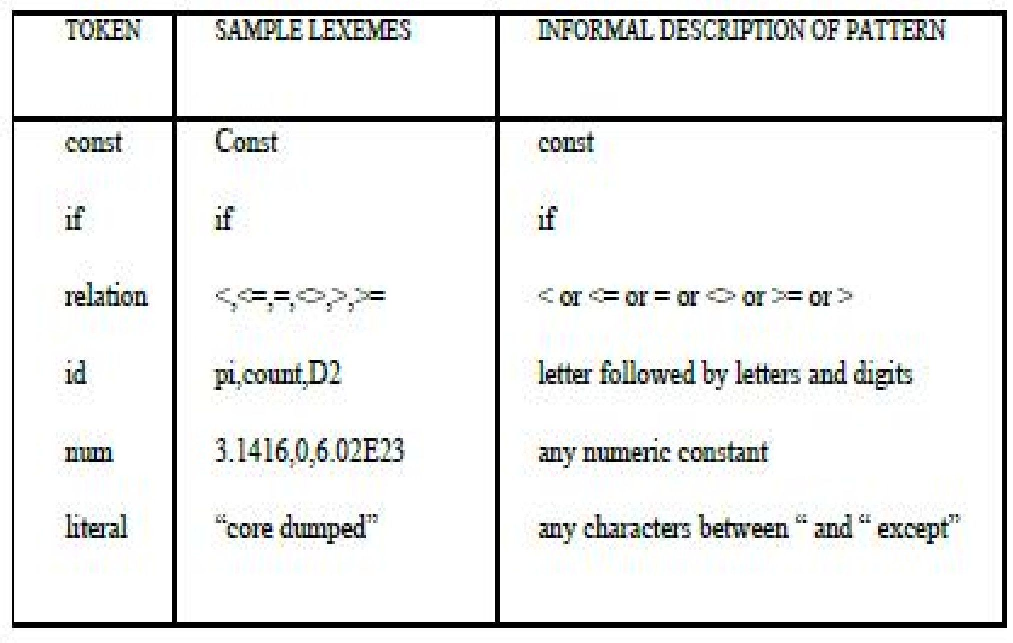
Tokens, Patterns, Lexemes

- **Token - pair of:**
  - token name – abstract symbol representing a kind of lexical unit
    - keyword, identifier, …
  - optional attribute value
  - The process of forming tokens from an input stream of characters is called **tokenization**
- **Pattern**
  - It is the description of the form that the lexeme of a token may take.
  - example
    - for a keyword the pattern is the character sequence forming that keyword.
    - for identifiers the pattern is a complex structure that is matched by many strings.
- **Lexeme**
  - It is a sequence of characters in the source program matching a pattern for a token.(or Collection of characters forming tokens is called Lexeme.)

Attributes for Tokens

- More than one lexeme can match a pattern
- Token number matches 0, 1, 100, 77,…
- Lexical analyzer must return
  - Not only the token name
  - Also an attribute value describing the lexeme represented by the token
- Token id may have associated information like
  - Lexeme
  - Type

    - Location – in order to issue error messages
- Token id attribute
    - Pointer to the symbol table for that identifier



TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	Const	const
if	if	if
relation	<,<=,=,<>,>,>=	< or <= or = or <> or >= or >
id	pi,count,D2	letter followed by letters and digits
num	3.1416,0,6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except"

Table 1.3: Token , lexeme and patterns

<u>Possible error recovery actions in a lexical analyzer:</u>

i. Panic mode recovery.

Delete successive characters from the remaining input until the analyzer can find a well-formed token.

May confuse the parser.

ii. Possible error recovery actions.

Deleting or Inserting input characters.

Replacing or transposing characters.

THE INPUT BUFFERING:

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Figure1.14 shows a buffer divided into two halves of, say 100 characters each.

One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

We often have to look one or more characters beyond the next lexeme before we can besure we have the right lexeme. As characters are read from left to right, each character is storedin the buffer to form a meaningful token as shown below

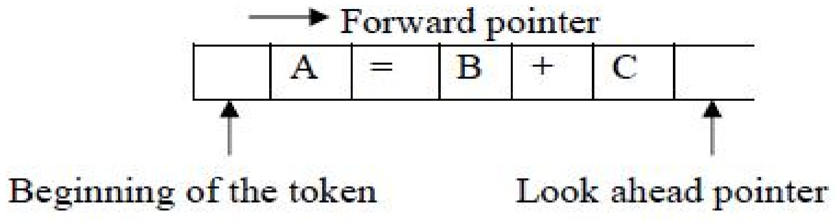


Fig 1.11 : input buffer

We introduce a two-buffer scheme that handles large look ahead safely. We thenconsider an improvement involving "sentinels" that saves time checking for the ends of buffers.

1.9.1 BUFFER PAIRS

A buffer is divided into two N-character halves, as shown below.

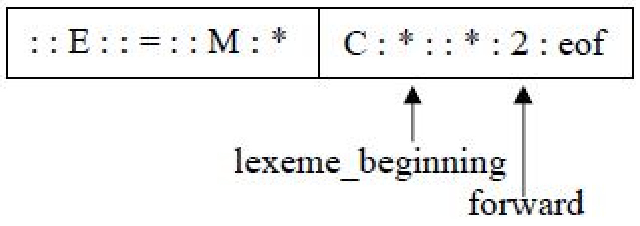


Fig 1.12 Buffer Pairs

- Each buffer is of the same size N, and N is usually the number of characters on one diskblock. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, representedby **eof**, marks the end of the source file.
- Two pointers to the input are maintained:
- Pointer **lexeme_ beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine Pointer **forward** scans ahead until a pattern match is found.
- Once the next lexeme is determined, forward is set to the character at its right end.
- The string of characters between the two pointers is the current lexeme.
- After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_ beginning is set to the character immediately after the lexeme just found.

Advancing forward pointer:

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

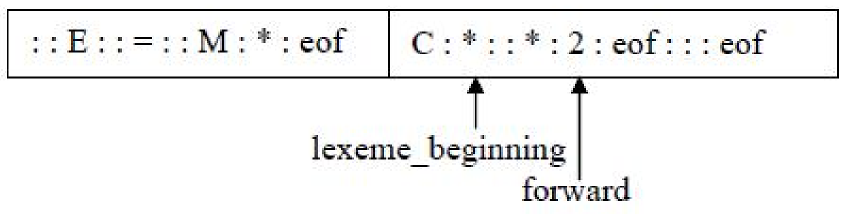
Code to advance forward pointer:

```
if forward at end of first half then begin
reload second half;
forward := forward + 1
end
else if forward at end of second half then begin
reload second half;
move forward to beginning of first half
end
else forward := forward + 1;
```

SENTINELS:

For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character of.

The sentinel arrangement is as shown below:



Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

Fig 1.13 sentinel arrangement

Code to advance forward pointer:

```
forward : = forward + 1;
if forward ↑ = eof then begin
if forward at end of first half then begin
reload second half;
forward := forward + 1
end
else if forward at end of second half then begin
reload first half;
move forward to beginning of first half
end
else /* eof within a buffer signifying end of input */
terminate lexical analysis
end
```

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:
1) Strings
2) Language
3) Regular expression

Strings and Languages
An **alphabet** or character class is a finite set of symbols.
A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
A **language** is any countable set of strings over some fixed alphabet.
In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s.
For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings
The following string-related terms are commonly used:
1. A **prefix** of string's' is any string obtained by removing zero or more symbols from the end of strings. For example, ban is a prefix of banana.
2. A **suffix** of string's' is any string obtained by removing zero or more symbols from the beginning of ' s'. For example, nana is a suffix of banana.
3. A **substring** of s is obtained by deleting any prefix and any suffix forms. For example, nan is a substring of banana.
4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of's'. For example, baan is a subsequence of banana.

Operations on languages:
The following are the operations that can be applied to languages:
1.Union
2.Concatenation
3.Kleene closure
4.Positive closure
The following example shows the operations on strings:
Let L={0,1} and S={a,b,c}
1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : L^*={ ε,0,1,00….}
4. Positive closure : L^+={0,1,00….}

Regular Expressions
Each regular expression r denotes a language L(r).
Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and $L(\varepsilon)$ is { ε }, that is, the language whose sole member is the empty string.
2. If ‘a’ is a symbol in Σ, then ‘a’ is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with ‘a’ in its one position.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,
a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
b) (r)(s) is a regular expression denoting the language L(r)L(s).
c) (r)* is a regular expression denoting (L(r))*.
d) (r) is a regular expression denoting L(r).
4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

Regular set
A language that can be defined by a regular expression is called a regular set.
If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.
There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.
For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions
Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form
$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
………
$d_n \rightarrow r_n$
1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet Σ U {d_1, d_2,. . . , d_{i-1}}.
Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:
letter → A | B | …. | Z | a | b | …. | z |
digit → 0 | 1 | …. | 9
id → letter (letter | digit) *

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. *One or more instances (+)*:
- The unary postfix operator + means “ one or more instances of” .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a’s.

- The operator + has the same precedence and associativity as the operator *.

2. *Zero or one instance (?)*:
- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. *Character Classes*:
- The notation [abc] where a, b and c are alphabet symbols denotes the regular expressiona | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression,[A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

- $ab^*|c$ means $(a(b)^*)|(c)$

- **Ex:**
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon,0,00,000,0000,....\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

RECOGNITION OF TOKENS
Consider the following grammar fragment:

stmt → if expr then stmt
| if expr then stmt else stmt
| ε
expr → term relop term
| term
term → id
| num

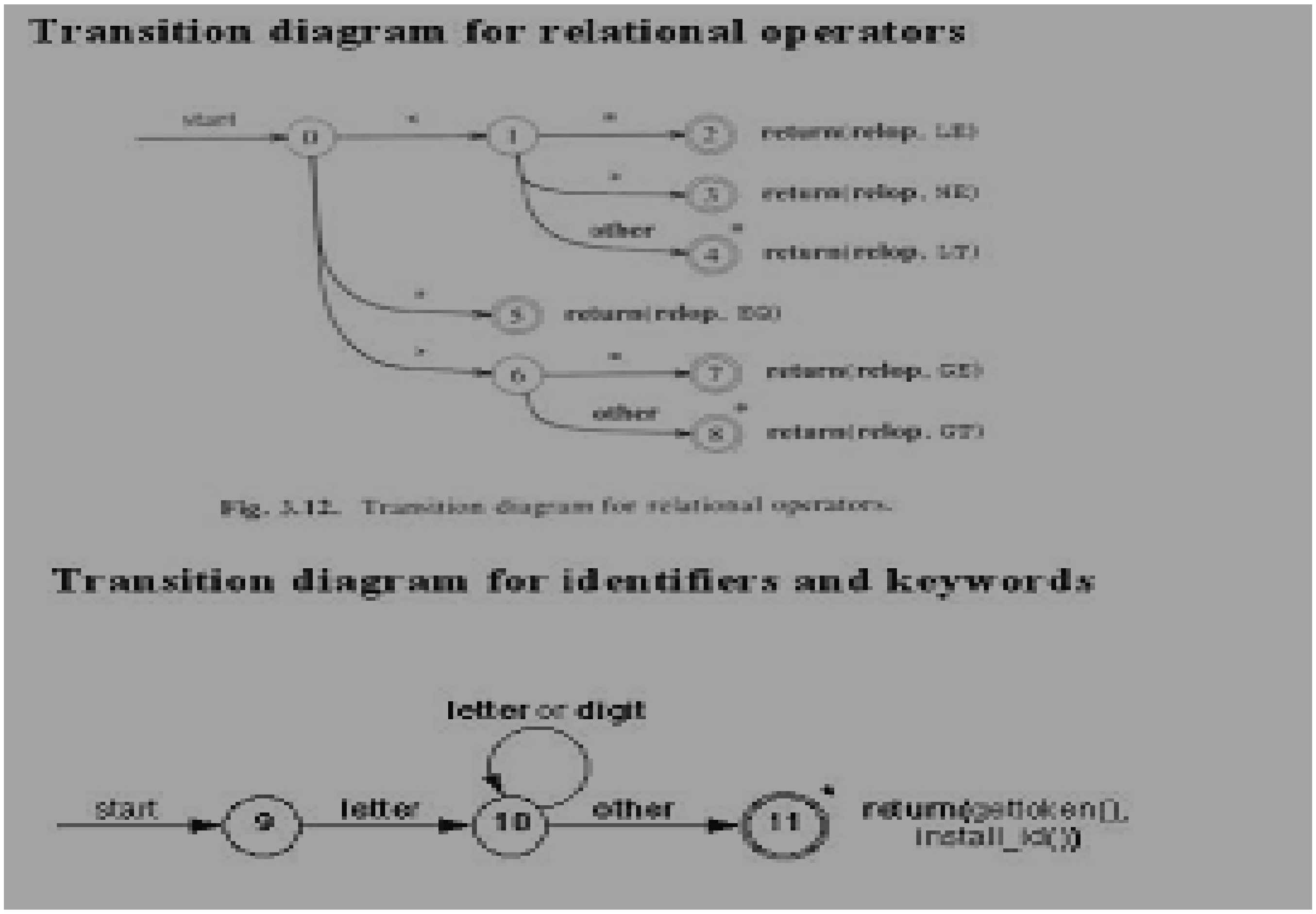
where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:
if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter(letter|digit)
*

num → digit^{+} $(.\text{digit}^{+})?(E(+|-)?\text{digit}^{+})?$

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

Transition diagrams

It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

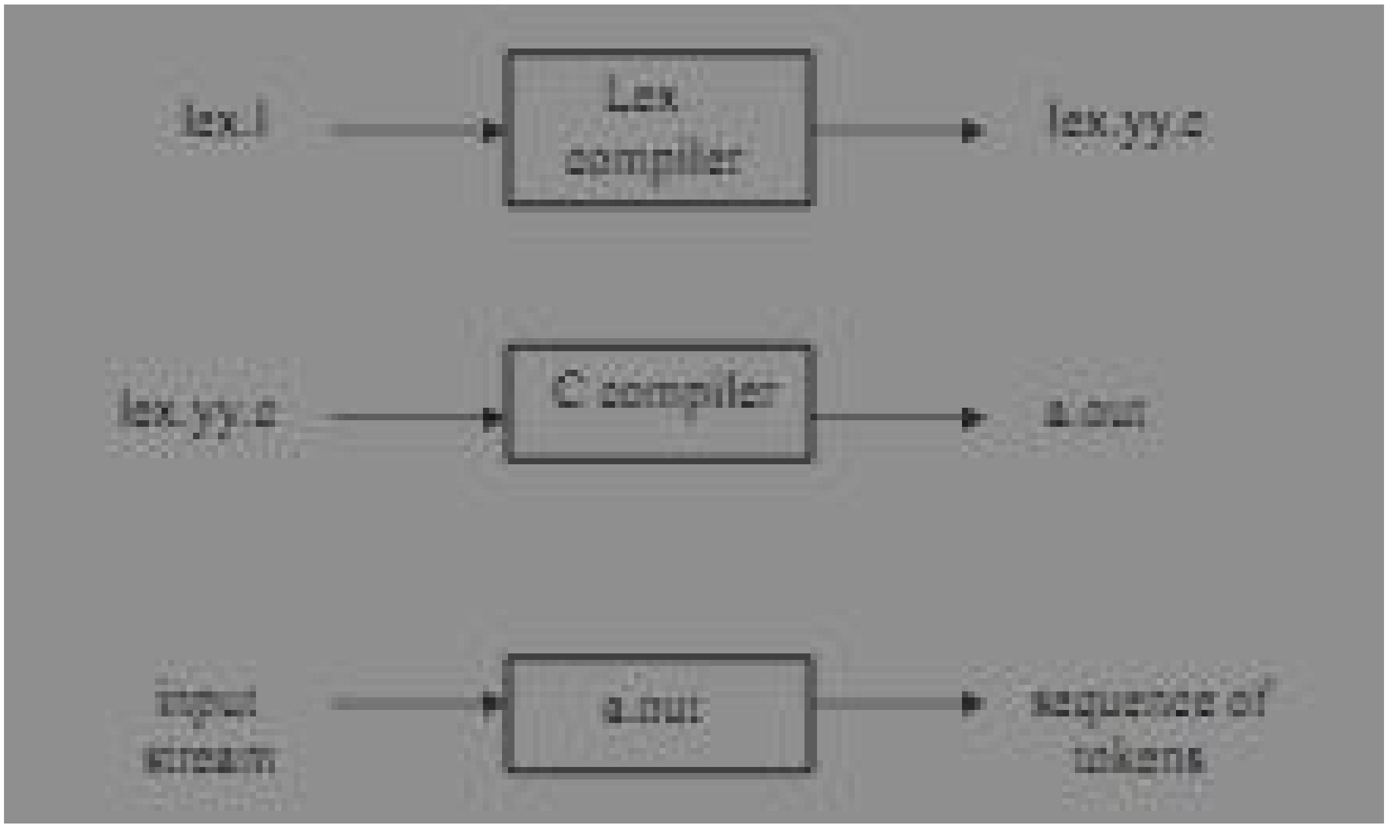

Transition diagram for relational operators
start
other
return(relop, LE)
return(relop, NE)
return(relop, LT)
return(relop, EQ)
return(relop, GE)
return(relop, GT)
Fig. 3.12. Transition diagram for relational operators.
Transition diagram for identifiers and keywords
letter or digit
start
letter
other
return(gettoken(), install_id())

A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- Lex
- YACC
- LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.
Creating a lexical analyzer
 First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
 Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



Lex Specification
A Lex program consists of three parts:
{ definitions }
%%
{ rules }
%%
{ user subroutines }

- Definitions include declarations of variables, constants, and regular definitions
- Rules are statements of the form

```
p1 {action1}
p2 {action2}
…
pn {actionn}
```

where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pimatches a lexeme. Actions are written in C code.

- User subroutines are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

YACC- YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

FINITE AUTOMATA

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

Types of Finite Automata

There are two types of Finite Automata :

i. Non-deterministic Finite Automata (NFA)
ii. Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by
M = {Qn, Σ, δ, q0, fn}
Qn – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q0 – starting state
fn – final state

Deterministic Finite Automata

DFA is a special case of a NFA in which
i) no state has an ε-transition.
ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

M = {Qd, Σ, δ, q0, fd}
Qd – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q0 – starting state
fd – final state

Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:
i) Convert RE to NFA using Thomson's rules
ii) Convert NFA to DFA
iii) Construct minimized DFA