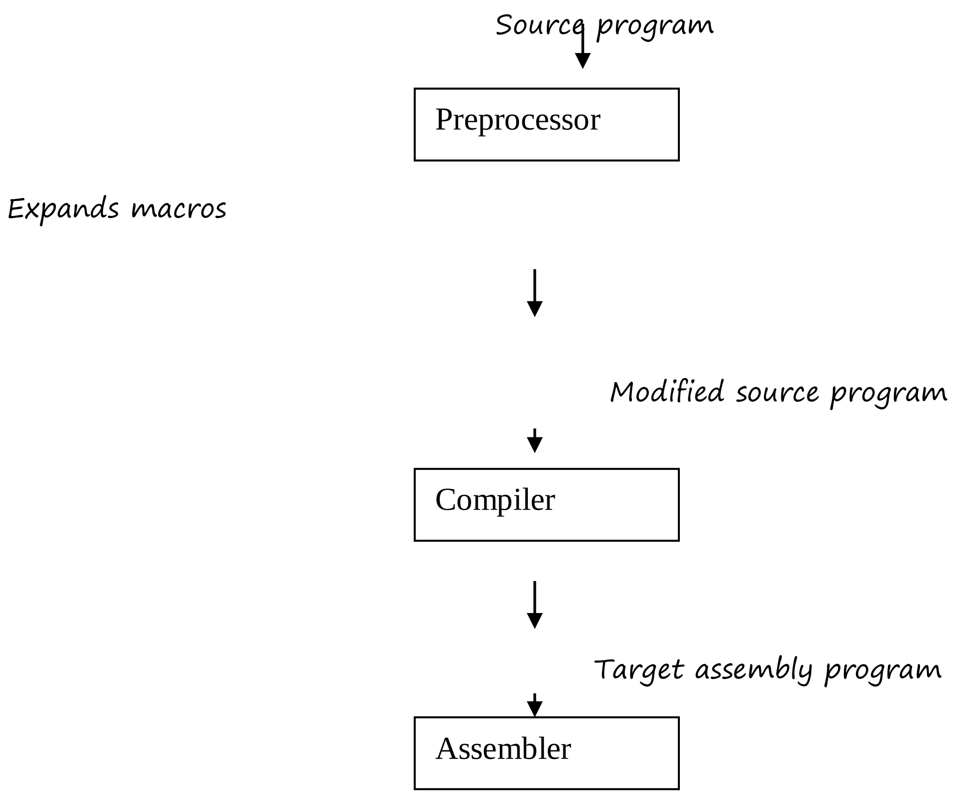
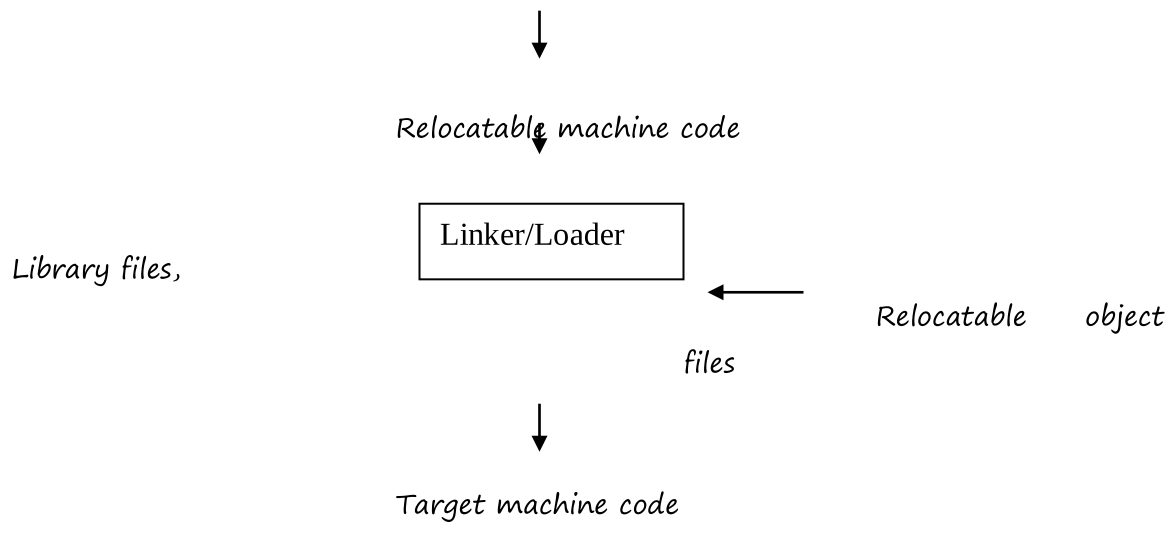
1. Introduction

Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer. Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.

Who developed the first compiler for a computer programming language?
The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term compiler was coined by Hopper. The A-0 functioned more as a loader or linker than the modern notion of a compiler. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957.

Language Processing System





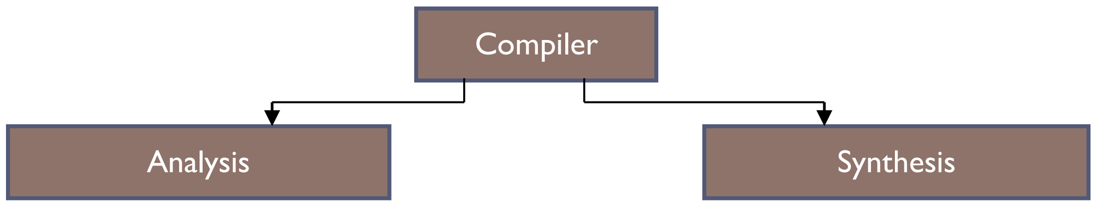
Pre-processor, Assembler, Loader and linker are also known as cousins of compiler.

Compilers and Interpreters

Compilers generate machine code, whereas interpreters interpret intermediate code Interpreters are easier to write and can provide better error messages (symbol table is still available) Interpreters are at least 5 times slower than machine code generated by compilers Interpreters also require much more memory than machine code generated by compilers Examples: Perl, Python, Unix Shell, Java, BASIC, LISP.

The Structure of a compiler

A compiler is a huge program that can consists of 10,000 to 1,000,000 lines of code. Since compiler is a huge program it is difficult to understand the entire compilation process. So the process is divided into number of modules called as PHASES. There are two major phases of compiler.



Analysis of the source program
Synthesis of a machine-language program

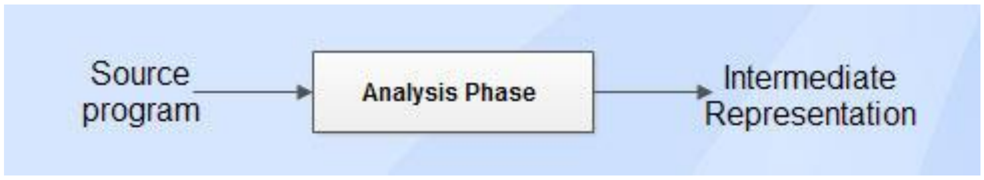
The structure of compiler consists of two parts:

Analysis part

• Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.

• It is also termed as front end of compiler.

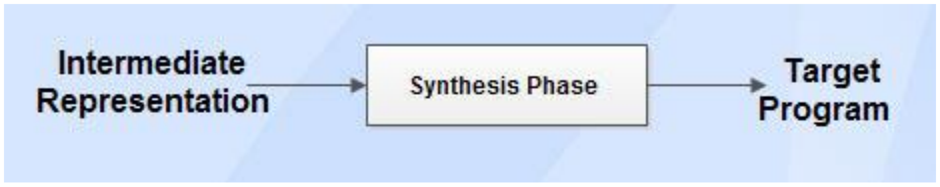
• Information about the source program is collected and stored in a data structure called symbol table.



Synthesis part

• Synthesis part takes the intermediate representation as input and transforms it to the target program.

• It is also termed as back end of compiler.



The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

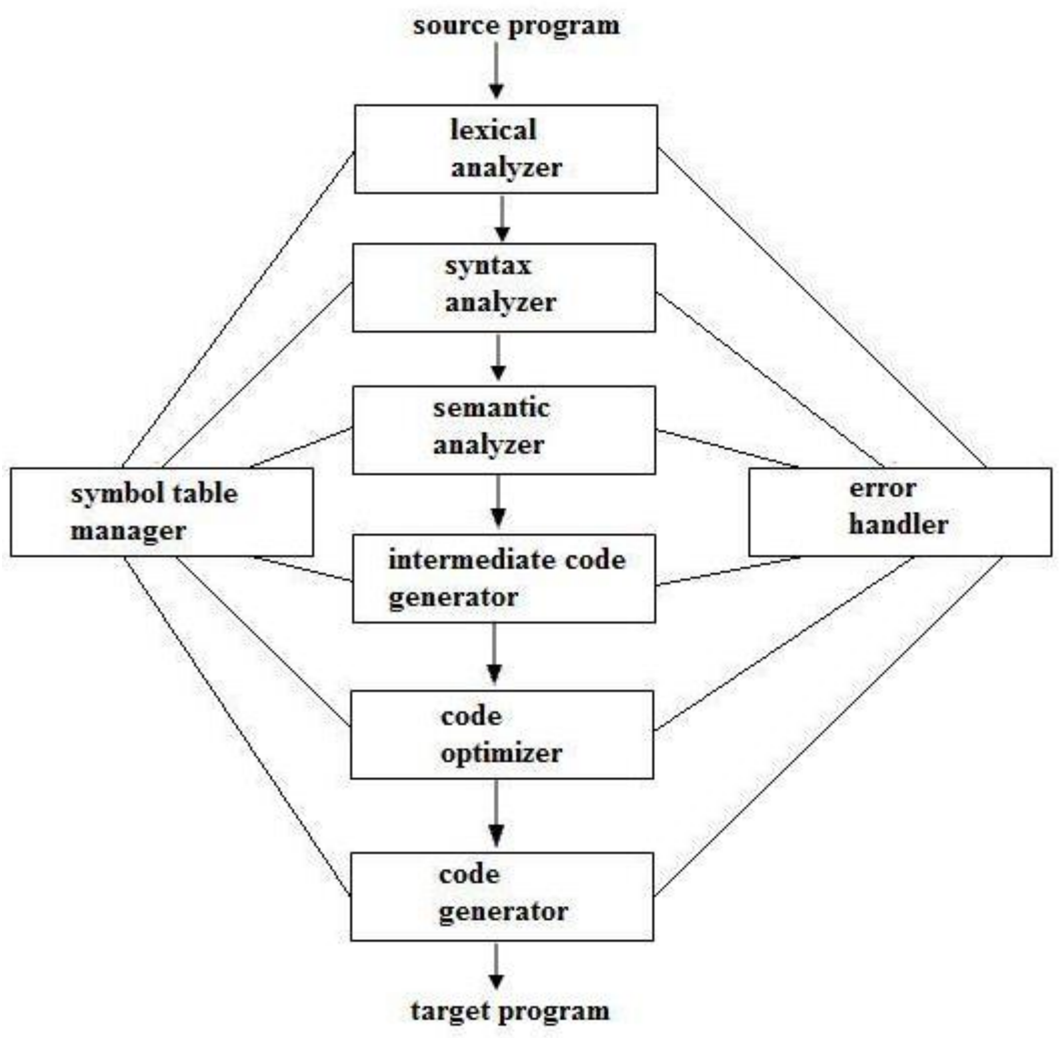


Fig 1.1 Structure of Compiler

The phases of compiler are as follows:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

All of the aforementioned phases involve the following tasks:

• Symbol table management.

• Error handling.

Lexical Analysis

• Lexical analysis is the first phase of compiler which is also termed as scanning.

• Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

• **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,

• **Pattern:** Pattern describes the rule that the lexemes of a token takes. It is the structure that must be matched by strings.

• Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token

Token Template: <token-name, attribute-value>

(eg.) c=a+b*5;

Lexemes and tokens

Lexemes	**Tokens**
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)

b	identifier
*	* (multiplication symbol)
5	5 (number)

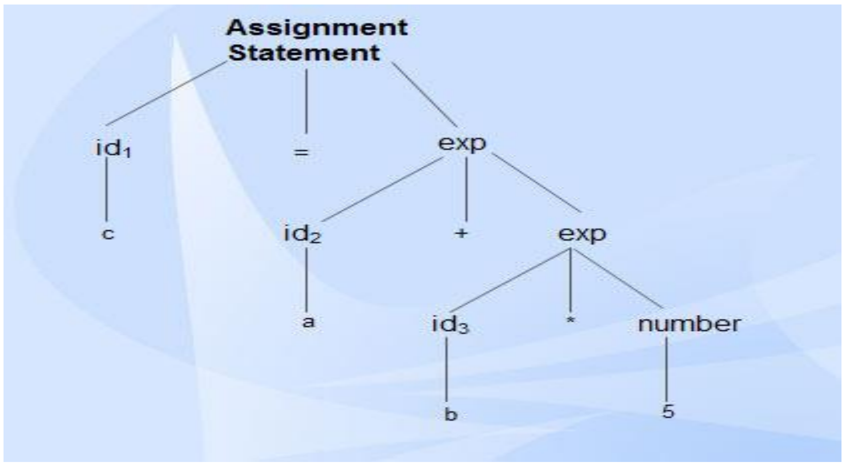
Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

Syntax Analysis

• Syntax analysis is the second phase of compiler which is also called as parsing.

• Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.

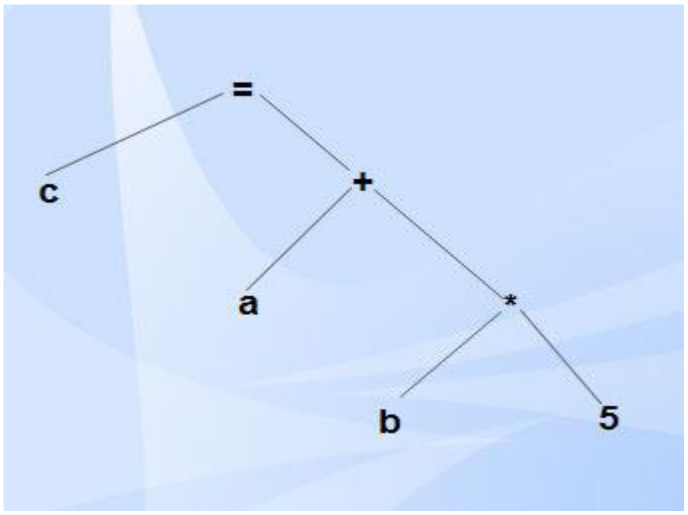
• A parse tree describes the syntactic structure of the input.



• Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

Input: Tokens

Output: Syntax tree



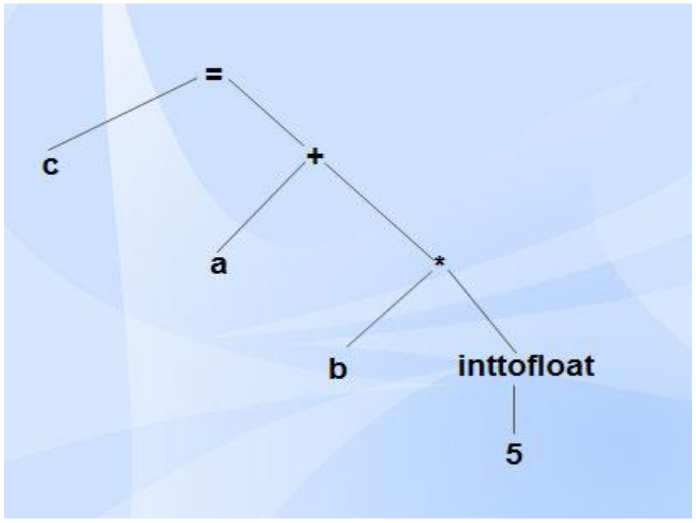
Semantic Analysis

• Semantic analysis is the third phase of compiler.

• It checks for the semantic consistency.

• Type information is gathered and stored in symbol table or in syntax tree.

• Performs type checking.



Intermediate Code Generation

• Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- o Postfix notation
- o Three address code

o Syntax tree

Most commonly used form is the three address code.

t_1 = inttofloat (5)

t_2 = id_3* tl

t_3 = id_2 + t_2

id_1 = t_3

<u>Properties of intermediate code</u>

- It should be easy to produce.
- It should be easy to translate into target program.

Code Optimization

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It results in faster running machine code.
- It can be done by reducing the number of lines of code for a program.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves

    o Deduction and removal of dead code (unreachable code).

    o Calculation of constants in expressions and terms.

    o Collapsing of repeated expression into temporary string.

    o Loop unrolling.

    o Moving code outside the loop.

    o Removal of unwanted temporary variables.

t_1 = id_3* 5.0

id_1 = id_2 + t_1

Code Generation

• Code generation is the final phase of a compiler.

• It gets input from code optimization phase and produces the target code or object code as result.

• Intermediate instructions are translated into a sequence of machine instructions that perform the same task.

• The code generation involves

  o Allocation of register and memory.

  o Generation of correct references.

  o Generation of correct data types.

  o Generation of missing code.

LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1

Symbol Table Management

• Symbol table is used to store all the information about identifiers used in the program.

• It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

• It allows finding the record for each identifier quickly and to store or retrieve data from that record.

• Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Example

int a, b; float c; char z;

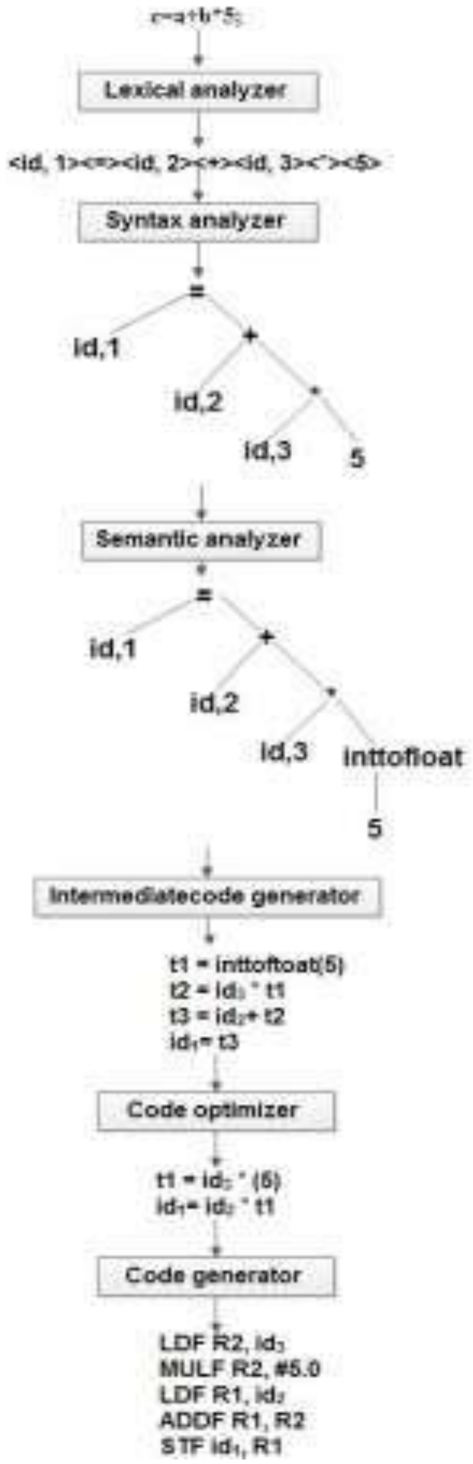
Symbol name	**Type**	**Address**
a	Int	1000

b	Int	1002
c	Float	1004
z	char	1008

Example

```
extern double test (double x);
  double sample (int count)
{
  double sum= 0.0;
    for (int i = 1; i < = count; i++)
      sum+= test((double) i);
      return sum;
  }
```

Symbol name	**Type**	**Scope**
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement



```
t1 = inttofloat(5)
t2 = id3 * t1
t3 = id2+ t2
id1= t3
```

```
t1 = id3 * (5)
id1= id2 * t1
```

```
LDF R2, id3
MULF R2, #5.0
LDF R1, id2
ADDF R1, R2
STF id1, R1
```

Error Handling

• Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =a+ b * 5.

Lexical Analysis

Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens.

A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

• Scanning

• Tokenization

Token, Pattern and Lexeme

Token

Token is a valid sequence of characters which are given by lexeme. In a programming language,

• keywords,

• constant,

• identifiers,

• numbers,

• operators and

• punctuations symbols

are possible tokens to be identified.

Pattern

Pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.

Lexeme

Lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a

token.

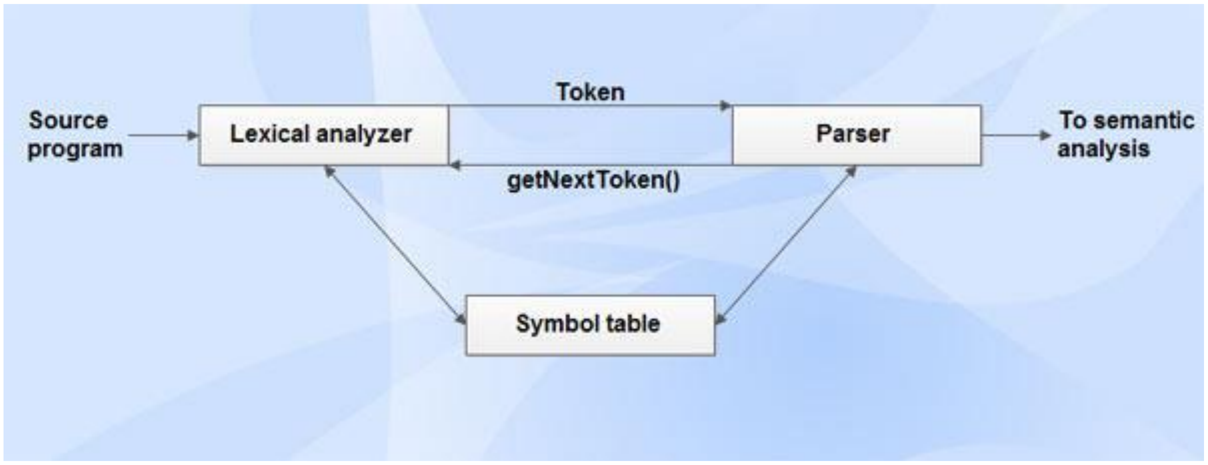
(eg.) c=a+b*5;

Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of Lexical Analyzer



Lexical analyzer performs the following tasks:

• Reads the source program, scans the input characters, group them into lexemes and produce the token as output.

• Enters the identified token into the symbol table.

• Strips out white spaces and comments from source program.

• Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.

• Expands the macros if it is found in the source program.

Tasks of lexical analyzer can be divided into two processes:

Scanning: Performs reading of input characters, removal of white spaces and comments.

Lexical Analysis: Produce tokens as the output.

Need of Lexical Analyzer

Simplicity of design of compiler The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.

Compiler efficiency is improved Specialized buffering techniques for reading characters speed up the compiler process.

Compiler portability is enhanced

Running example:

float abs_zero_Kelvin = -273;

Token (also called word)

A string of characters which logically belong together float, identifier, equal, minus, intnum, semicolon Tokens are treated as terminal symbols of the grammar specifying the source language

Pattern

The set of strings for which the same token is produced The pattern is said to match each string in the set

float, l(l+d+_)*, =, -, d+, ;

Lexeme

The sequence of characters matched by a pattern to form the corresponding token

“float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

Issues in Lexical Analysis

Lexical analysis is the process of producing tokens from the source program. It has the following issues:

• Lookahead

• Ambiguities

Lookahead

Lookahead is required to decide when one token will end and the next token will begin. The simple example which has lookahead issues are i *vs. if,* = *vs.* ==. Therefore a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

• Is if it is two variables *i* and *f* or if?

• Is == is two equal signs =, = or ==?

• arr(5, 4) vs. fn(5, 4) *II* in Ada (as array reference syntax and function call syntax are similar.

Hence, the number of lookahead to be considered and a way to describe the lexemes of each token is also needed.

Regular expressions are one of the most popular ways of representing tokens.

Ambiguities

The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

• The longest match is preferred.

• Among rules which matched the same number of characters, the rule given first is preferred.

Lexical Errors

• A character sequence that cannot be scanned into any valid token is a lexical error.

• Lexical errors are uncommon, but they still must be handled by a scanner.

• Misspelling of identifiers, keyword, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Error Recovery Schemes

- Panic mode recovery
- Local correction
  - Source text is changed around the error point in order to get a correct text.
  - Analyzer will be restarted with the resultant new text as input.
- Global correction
  - It is an enhanced panic mode recovery.
  - Preferred when local correction fails.

Panic mode recovery

In panic mode recovery, unmatched patterns are deleted from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

(eg.) For instance the string fi is encountered for the first time in a C program in the context:

fi (a== f(x))

A lexical analyzer cannot tell whether *f* iis a misspelling of the keyword if or an undeclared function identifier.

Since *f* i is a valid lexeme for the token **id,** the lexical analyzer will return the token **id** to the parser.

Lexical error handling approaches

Lexical errors can be handled by the following actions:

- Deleting one character from the remaining input.
- Inserting a missing character into the remaining input.
- Replacing a character by another character.
- Transposing two adjacent characters.

Languages
Symbol: An abstract entity, not defined
Examples: letters and digits
String: A finite sequence of juxtaposed symbols
abcb, caba are strings over the symbols a,b, and c
|w| is the length of the string w, and is the #symbols in it
  is the empty string and is of length 0
Alphabet: A finite set of symbols
Language: A set of strings of symbols from some alphabet
Φ and {□} are languages
The set of palindromes over {0,1} is an infinite language
The set of strings, {01, 10, 111} over {0,1} is a finite language
If Σ is an alphabet, Σ ∗ is the set of all strings over Σ

Example: A *scanner* groups input characters into tokens. For example, if the input is

```
x = x*(b+1);
```

then the scanner generates the following sequence of tokens:

```
id(1)
=
id(1)
*
(
id(2)
+
num(1)
)
;
```

where `id(1)` indicates the identifier with name `x` (a program variable in this case) and `num(1)` indicates the integer `1`. Each time the parser needs a token, it sends a request to the scanner. Then, the scanner reads as many characters from the input stream as it is necessary to construct a single token. The scanner may report an error during scanning (eg, when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the scanner is suspended and returns the token to the parser. The parser will repeatedly call the scanner to read all the tokens from the input stream or until an error is detected (such as a syntax error).

Tokens are typically represented by numbers. For example, the token `*` may be assigned the number 35. Some tokens require some extra information. For example, an identifier is a token (so it is represented by some number) but it is also associated with a string that holds the identifier name. For example, the token `id(x)` is associated with the string, `"x"`. Similarly, the token `num(1)` is associated with the number, 1.

Specification of Tokens

Tokens are specified by patterns, called *regular expressions*. For example, the regular expression `[a-z][a-zA-Z0-9]*` recognizes all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.

Describe the languages denoted by the following regular expressions:

1. a(a|b)*a
2. ((ε|a)b*)*
3. (a|b)*a(a|b)(a|b)
4. a*ba*ba*ba*
5. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Answer

1. String of a's and b's that start and end with a.
2. String of a's and b's.
3. String of a's and b's that the character third from the last is a.
4. String of a's and b's that only contains three b.
5. String of a's and b's that has a even number of a and b.

Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

Answer

```
select -> [Ss][Ee][Ll][Ee][Cc][Tt]
```

Write regular definitions for the following languages:

1. All strings of lowercase letters that contain the five vowels in order.
2. All strings of lowercase letters in which the letters are in ascending lexicographic order.
3. Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (")
4. All strings of a's and b's that do not contain the substring abb.
5. All strings of a's and b's that do not contain the subsequence abb.

Answer

1、

```
want -> other* a (other|a)* e (other|e)* i (other|i)* o (other|o)* u (other|u)*
other -> [bcdfghjklmnpqrstvwxyz]
```

2、

```
a* b* ... z*
```

3、

```
\/\*([^*"]*|".*"|\*+[^/])*\*\/
```

4

```
b*(a+b?)*
```

5、

```
b* | b*a+ | b*a+ba*
```

Write character classes for the following sets of characters:

1. The first ten letters (up to "j") in either upper or lower case.
2. The lowercase consonants.
3. The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
4. The characters that can appear at the end of alegitimate English sentence (e.g. , exclamation point) .

Answer

1. [A-Ja-j]
2. [bcdfghjklmnpqrstvwxzy]
3. [0-9a-f]
4. [.?!]

Note that these regular expressions give all of the following symbols (operator characters) a special meaning:

```
\ " . ^ $ [ ] * + ? { } | /
```

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression "**" matches the string ** . We can also get the literal meaning of an operator character by preceding it by a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Answer

```
\"\\
```

The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, ^[^aeiou]*$ matches any complete line that does not contain a lowercase vowel.

1. How do you tell which meaning of ^ is intended?

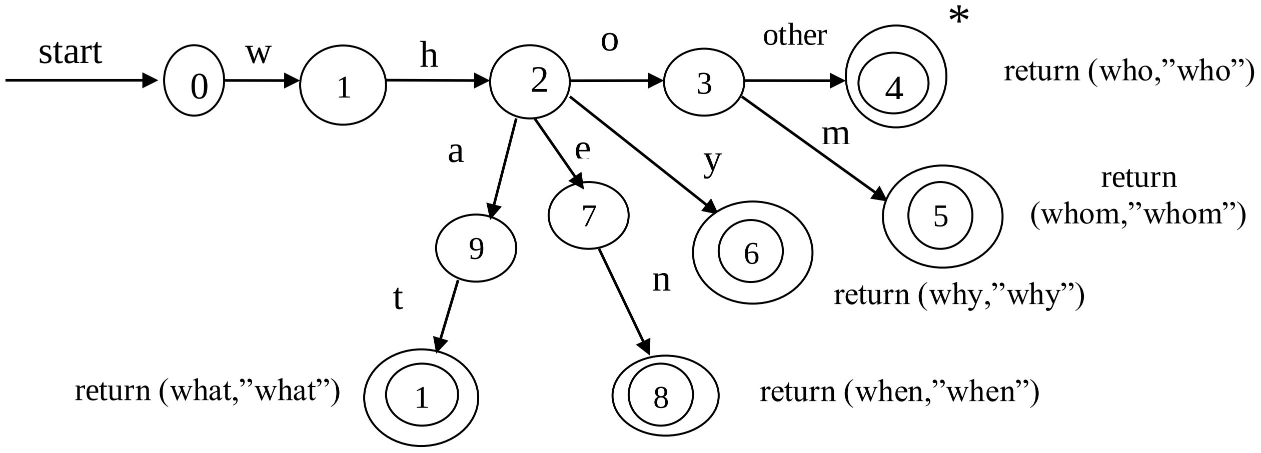
if ^ is in a pair of brakets, and it is the first letter, it means complemented classes, or it means the left end of a line.

Recognition of Tokens

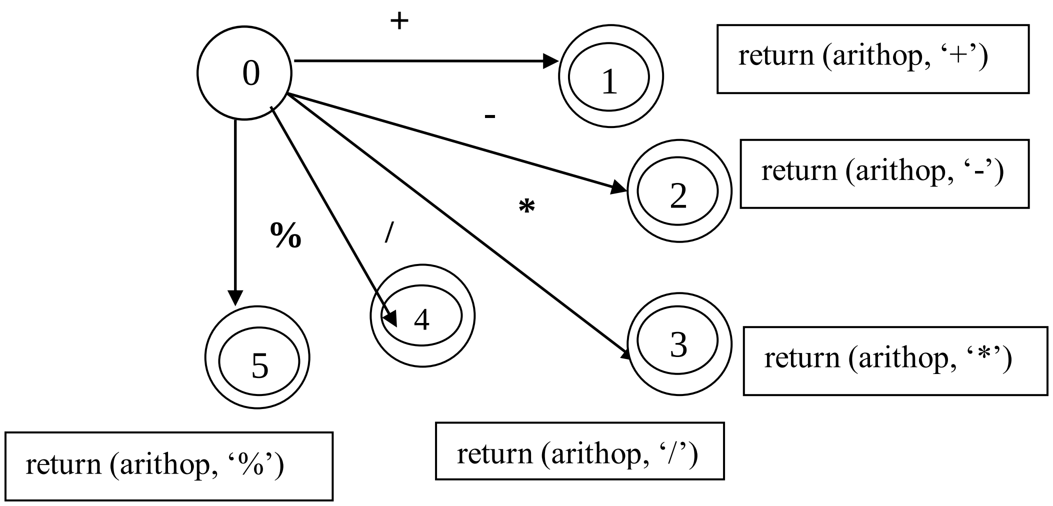
Construct transition diagram for the following

1. **who, when, what, why, whom**
2. **arithmetic operators**

who, when, what, why, whom



Arithmetic operators


0
+
1
return (arithop, '+')
-
2
return (arithop, '-')
*
3
return (arithop, '*')
/
4
return (arithop, '/')
%
5
return (arithop, '%')