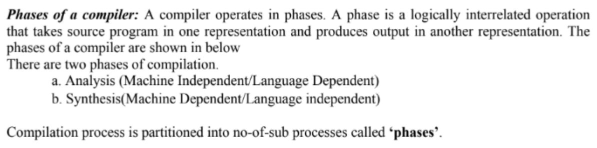
Q1. Explain different phases of compiler.

A compiler translates source code written in a high-level programming language into an equivalent machine code. Here are the phases of compilation process,


Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below
There are two phases of compilation.
a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

A. Analysis Phase

1. Lexical Analysis (Scanner)

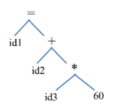
The initial phase of compilation scans the source code character by character to group them into tokens such as keywords, identifiers, constants, punctuation and operators, removing unnecessary whitespaces and comments. A symbol table is created to keep track of identifiers.

Position = initial + rate*60 $\longrightarrow$ id1 = id2 + id3 * 60

2. Syntax Analysis (Parser)

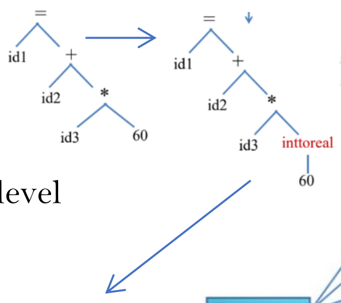
The second phase checks if the token sequence follows the grammatical rules, reporting syntax errors if violations occur. The parser constructs a parse tree.

id1 = id2 + id3 * 60 $\longrightarrow$



3. Semantic Analysis

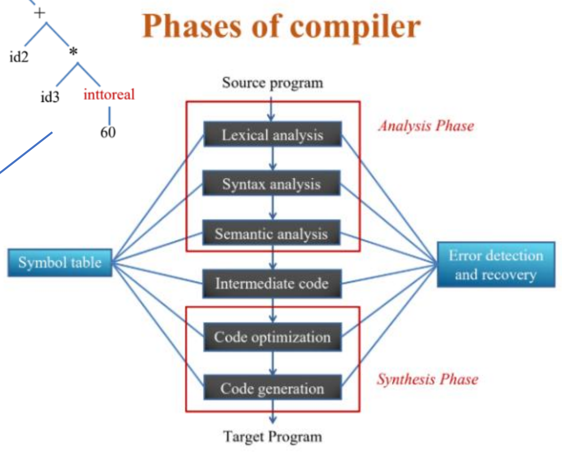
The semantic analysis phase ensures the program's logic is correct, checking for issues like undeclared variables or type mismatches.



4. Intermediate Code Generation

The intermediate code generator translates the parse tree into middle level three-address code, often resembling assembly language.

```
t1= int to real(60)
t2= id3 * t1
t3= id2 + t2
id1= t3
```



B. Synthesis Phase

5. Code Optimization

Improves the intermediate code by eliminating redundancy to make it more efficient without changing its behaviour.

```
t1= id3 * 60.0
id1 = id2 + t1
```

6. Code Generation

The final phase where the optimized code is translated into the target machine code or assembly language that can be executed on the computer.

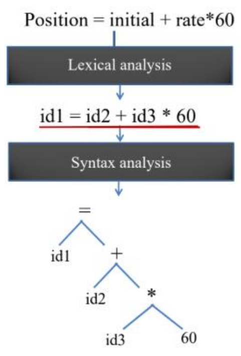
```
MOVF R2, id3
MULF R2, #60.0
MOVF R1, id2
ADDF R1,R2
MOVF id1, R1
```

Q2. Explain Semantic Analysis and Syntax Analysis phases of compiler with suitable example. Also explain the reporting errors by these two phases.

Syntax Analysis (Parser)

- **Purpose:** The second phase checks if the sequence of tokens generated by the lexical analyzer follows the grammatical rules of the source language. In other words if the program is syntactically correct or not. The parser builds a **parse tree** which represents the hierarchical structure of the program.
- **Process:** The parser examines the tokens to ensure that the source code follows the correct syntax. If the code violates the grammar, a syntax error is reported.

Error Reporting in Syntax Analysis:



- **Syntax errors** are reported when the tokens do not match the grammar of the language.
- **Example of Syntax Error:**

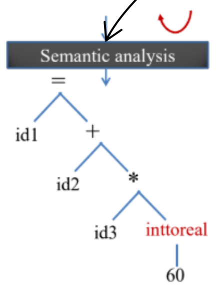
```
int a = ;  // Missing value after '='
```
Syntax Error: Missing expression after assignment.

```
int 5 = a; // Invalid variable name
```
Syntax Error: Invalid identifier '5'.

Semantic Analysis

- **Purpose:** This phase checks the meaning of the program, ensuring that it makes logical sense. It verifies that all operations are semantically correct, such as ensuring variables are declared before use, data types match and there are no logical contradictions.
- **Process:** During semantic analysis, the compiler checks the parse tree created by the syntax analyser, for errors like type mismatches or undeclared variables.

Error Reporting in Semantic Analysis:



- **Semantic errors** occur when the code is syntactically correct but logically flawed.
- **Example of Semantic Error:**

```
int x = 10;
y = x + z;  // Error: 'y' and 'z' are undeclared
```
Semantic Error: Undeclared variable 'y'.

```
int x = "hello";  // Error: Type mismatch, 'x' is an integer
```
Semantic Error: Cannot assign a string to an integer variable 'x'.

Q3. Write a short note on Symbol Table Management.

The symbol table is primarily created during the initial phase (lexical analysis phase) of compilation. It is a crucial data structure used by the compiler to store information about identifiers such as variables, functions, classes, etc. in the source program. It acts as a central repository to manage and retrieve information during the compilation process.

Purpose of Symbol Table

1. Store Identifiers: Stores information about identifiers such as variables, functions, classes, etc. that are used throughout the program.
2. Scope Management: Manages the scope of variables and functions, ensuring they are accessed only within their valid context.
3. Error Detection: Identifies undeclared variables, duplicate declarations, and other semantic errors.
4. Type Checking: Associates data types with identifiers and helps in type compatibility checks during semantic analysis.

Structure

A symbol table is often implemented as a hash table, tree or linked list. A symbol table contains the identifier name, data type, scope information, memory location, and additional attributes

Identifier	**Type**	**Scope**	**Memory Location**	**Attributes**
sum	float	Global	0x0034	-
arr	int[10]	Local	0x0050	Array size: 10

Operations on a Symbol Table

1. **Insertion**: Adds a new entry when an identifier is declared.
2. **Lookup**: Searches for an identifier to retrieve its details (e.g., type, scope).
3. **Modification**: Modifies attributes of existing entries (e.g., updating memory location).
4. **Deletion**: Removes identifiers when they go out of scope.

Symbol Table Management in Compiler Phases:

- During the **Lexical Analysis phase** identifiers are added to the symbol table.
- In **Syntax Analysis phase,** the parser uses the table to verify structure. And **Semantic Analyser** performs type and scope checking using the table.

- **Intermediate Code Generation** retrieves memory locations and attributes for code translation from symbol table.

Q4. What is the pass of a compiler? Explain how the single and multi-pass compilers work.

A **pass** in a compiler refers to a complete traversal of the source program or its intermediate representation to perform specific tasks. A compiler may make one or more passes over the source code depending on its design.

Single-Pass Compiler

- **Definition:** A **single-pass compiler** processes the source code in one go. It reads the code, analyses it, generates an intermediate representation and produces the final output in a single pass through the code.
- **How it Works**: Each phase of the compilation (lexical analysis, syntax analysis, semantic analysis and code generation) is executed in a single pass.
- **Advantages**: Faster compilation as it processes the code in one go. Requires less memory since intermediate results are not stored.
- **Disadvantages**: Limited optimization capabilities and cannot handle forward references

  Example: Early compilers like Pascal.

Multi-Pass Compiler

- **Definition:** A **multi-pass compiler** processes the source code in multiple passes. Each pass performs a specific task, such as lexical analysis, syntax analysis, optimization, etc., and can access information gathered in previous passes.
- **How It Works:**
    - **First Pass:** The compiler gathers all necessary information (e.g., building the symbol table).
    - **Subsequent Passes:** The compiler refine or optimize the intermediate representation and eventually generate machine code.
- **Advantages**: More flexible, capable of performing complex optimizations and handling more intricate source code structures.
- **Disadvantages**: Slower compilation time and requires more memory due to multiple passes through the code.

  Example: Modern compilers like GCC and LLVM.

Q5. List out phases of a compiles. Write a brief note on Lexical Analyzer.

Phases of a Compiler:

1. **Lexical Analysis**
2. **Syntax Analysis**
3. **Semantic Analysis**
4. **Intermediate Code Generation**
5. **Code Optimization**
6. **Code Generation**

Lexical Analyzer (Scanner)

Purpose:

The lexical analyzer, also known as the scanner, is the first phase of the compiler. It is responsible for reading the source code character by character and converting it into a sequence of tokens, which are the smallest units of meaningful code like keywords, operators, identifiers, constants and punctuations.

Function:

- Tokenization: It breaks down the source code into recognizable units (tokens) such as keywords (int, if), operators (+, -), identifiers (variable names like x), constants (10, 3.14), and punctuation (semicolons, parentheses).
- Eliminating Whitespace and Comments: It removes unnecessary characters such as spaces, tabs, and comments from the code, which do not contribute to the program's execution.
- Error Handling: It detects invalid characters or malformed and reports lexical errors.

Example: For the statement int x = 5;, the tokens might be: int, x, =, 5, ;.

Q6. How do the parser and scanner communicate? Explain with the block diagram communication between them. Also explain: What is input buffering?

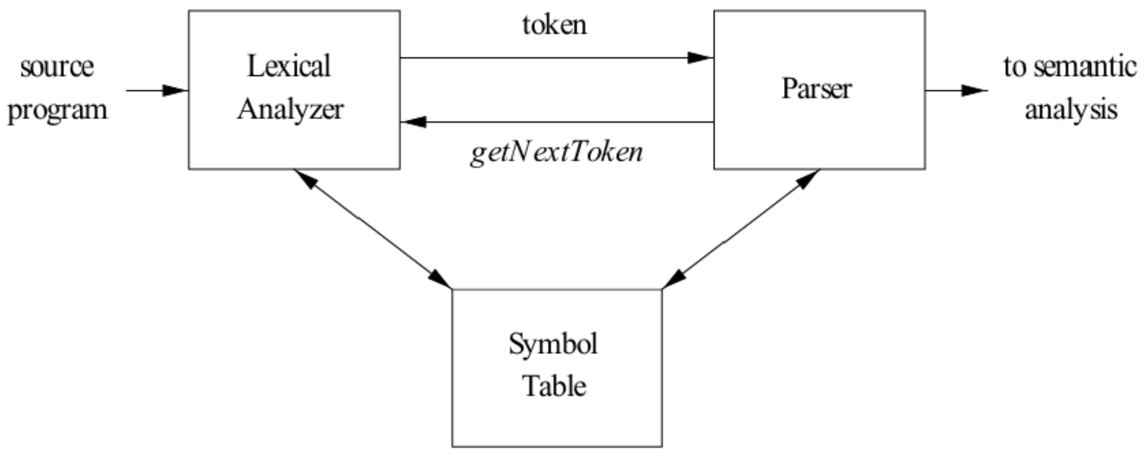
Communication Between Parser and Scanner

The **parser** and **scanner** are two key components of a compiler, and they communicate during the **lexical analysis** and **syntax analysis** phases of compilation. Here's how they work together:

- **Scanner (Lexical Analyzer):** The scanner reads the source code character by character, breaking it into **tokens**. The scanner's job is to convert the raw source code into a structured format that the parser can understand.
- **Parser (Syntax Analyzer):** The parser takes the sequence of tokens generated by the scanner and arranges them to create a **syntax tree**. The parser checks if the tokens form valid constructs according to the language's syntax rules.

Communication Process: Upon receiving a "Get next token" command from parser, the **scanner** feeds tokens to the **parser** one by one. The **parser** uses these tokens to check the structure of the code and build the parse tree. If the parser encounters a syntactic error, it will request the scanner to provide additional tokens.

Block Diagram:



Input Buffering

Input buffering is a technique used to improve the efficiency of the **scanner** (lexical analyzer) while reading the source code. It reduces the number of disk or memory accesses by storing chunks of the input text in a buffer and feeding the scanner with this data in blocks instead of character-by-character reading.

Example of Input Buffering:

Consider a source code stream:

```
int x = 10;
```

The input buffer might store:

```
Buffer 1: "int x = "
Buffer 2: "10;"
```

Q7. Write the two methods used in lexical analyser for buffering the input. Which technique is used for speeding up the lexical analyser?

There are two main techniques for **input buffering** used in a **lexical analyser** to improve its efficiency while reading the input source code:

Buffer Pairs

- The input buffer is divided into two halves, each of size **N**, where NNN is the number of characters in a disk block.
- **Pointers**:
  - **Lexeme Begin**: Marks the start of the current lexeme.
  - **Forward**: Scans characters ahead to identify token boundaries.
- **How It Works**:
  1. When the forward pointer reaches the end of the first buffer, the second buffer is preloaded with the next NNN characters from the input.
  2. Similarly, when the forward pointer reaches the end of the second buffer, the first buffer is refilled.
  3. The process continues alternately between the two buffers.
- **Advantages**: Reduces the frequency of I/O operations by reading larger chunks of data.
- **Code to Advance Forward Pointer**:

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

2. Sentinels

- In this technique, a **sentinel character** (e.g., EOF) is added at the end of each buffer to simplify boundary checking.
- **How It Works**:
  1. Each buffer is extended to include a sentinel character that cannot appear in the source program.
  2. The forward pointer is advanced normally until it encounters the sentinel character (EOF).
  3. When EOF is encountered:
     - If the forward pointer is at the end of the first buffer, the second buffer is loaded.
     - If the forward pointer is at the end of the second buffer, the first buffer is reloaded.
     - If both buffers are exhausted, lexical analysis is terminated.

- **Advantages**: Eliminates the need for explicit checks for buffer boundaries, reducing two tests to one. Simplifies code and improves performance.
- **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 the beginning of the first half;
    end
    else terminate lexical analysis;
end
```

The **Sentinel Technique** is commonly used for speeding up the lexical analyzer because it reduces processing overhead by combining boundary and EOF checks into a single test, enabling faster and more efficient character scanning.

OR

The **Sentinel Technique** is commonly used for speeding up the lexical analyzer because it eliminates the need for separate boundary checks when the **Forward Pointer** reaches the end of a buffer. With a sentinel character (such as **EOF**) marking the end, the analyzer can immediately detect when it needs to reload a buffer, reducing overhead and improving efficiency.

Q9. Define the following terms: 1. Token 2. Pattern 3. Lexeme

Sequence of character having a collective meaning is known as token. In other word Tokens are the smallest units of meaningful code, processed by the parser. It is a **pair** consisting of a **token name**, represents the category of the token.

Examples of Tokens:

- Keywords (e.g., int, if), Operators (e.g., +, =), Identifiers (e.g., x, variableName), Constant (e.g., 10, "hello") and Punctuation (e.g., ;, {, })

Token Example: In the statement int x = 10;, the tokens are:

- int (Keyword), x (Identifier), = (Operator), 10 (Integer/Constant) and ; (Punctuation)

A **pattern** is a set of strings in the input for which the same token is produced as output. It defines the structure or format of valid lexemes for a particular token.

Patterns are often written in **regular expressions** that describe the valid format of tokens.

Examples of Patterns:

- A pattern for an **identifier** might be: `[a-z A-Z][a-z A-Z 0-9]*` (starts with a letter followed by letters or digits).
- A pattern for an **integer literal** might be: `[0-9]+` (one or more digits).

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example:

- **In the statement int x = 10;, the lexemes are:**
  - int (matches the pattern for the keyword token)
  - x (matches the pattern for the identifier token)
  - = (matches the pattern for the operator token)
  - 10 (matches the pattern for the integer literal token)
  - ; (matches the pattern for the punctuation token)

Q.10 Define lexeme, token and pattern. Identify the lexemes that make up the tokens in the following program segment. Indicate corresponding token and pattern.

(1)

```
void swap (int a, int b)
{
  int k;
  k = a;
  a = b;
  b = k;
}
```

Lexeme	Token Name	Pattern
`void`	**KEYWORD**	Exact match (`void`)
`swap`	**IDENTIFIER**	`[a-z A-Z][a-zA-Z0-9]*`
`(`	**PUNCTUATION**	Exact match (`(`)
`int`	**KEYWORD**	Exact match (`int`)
`a`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`,`	**PUNCTUATION**	Exact match (`,`)
`b`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`)`	**PUNCTUATION**	Exact match (`)`)
`{`	**PUNCTUATION**	Exact match (`{`)
`int`	**KEYWORD**	Exact match (`int`)
`k`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`;`	**PUNCTUATION**	Exact match (`;`)
`k`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`=`	**OPERATOR**	Exact match (`=`)
`a`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`

;	**PUNCTUATION**	Exact match (;)
a	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
=	**OPERATOR**	Exact match (=)
b	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
;	**PUNCTUATION**	Exact match (;)
b	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
=	**OPERATOR**	Exact match (=)
k	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
;	**PUNCTUATION**	Exact match (;)
}	**PUNCTUATION**	Exact match (})

Total Tokens Count for Segment (1): 26

(2)

```
String message = "Welcome to Java!";
System.out.println(message);
```

Lexeme	Token Name	Pattern
String	**KEYWORD**	Exact match (String)
message	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
=	**OPERATOR**	Exact match (=)
"Welcome to Java!"	**STRING_LITERAL**	"[^"]*"
System	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
.	**PUNCTUATION**	Exact match (.)
out	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
.	**PUNCTUATION**	Exact match (.)
println	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
(	**PUNCTUATION**	Exact match (()
message	**IDENTIFIER**	[a-zA-Z][a-zA-Z0-9]*
)	**PUNCTUATION**	Exact match ())
;	**PUNCTUATION**	Exact match (;)

Total Tokens Count for Segment (2): 13

(3) C Program Segment:

```
#include <stdio.h>
int main() {
  printf("Hello, C!\n");
  return 0;
}
```

Lexeme	Token Name	Pattern
#	**PUNCTUATION**	Exact match (#)
include	**KEYWORD**	Exact match (include)
<	**PUNCTUATION**	Exact match (<)
stdio.h	**HEADER_FILE**	[a-zA-Z0-9_.]+
>	**PUNCTUATION**	Exact match (>)
int	**KEYWORD**	Exact match (int)

`main`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`(`	**PUNCTUATION**	Exact match (`(`)
`)`	**PUNCTUATION**	Exact match (`)`)
`{`	**PUNCTUATION**	Exact match (`{`)
`printf`	**IDENTIFIER**	`[a-zA-Z][a-zA-Z0-9]*`
`(`	**PUNCTUATION**	Exact match (`(`)
`"Hello, C!\n"`	**STRING_LITERAL**	`"[^"]*"`
`)`	**PUNCTUATION**	Exact match (`)`)
`;`	**PUNCTUATION**	Exact match (`;`)
`return`	**KEYWORD**	Exact match (`return`)
`0`	**INTEGER_LITERAL**	`[0-9]+`
`;`	**PUNCTUATION**	Exact match (`;`)
`}`	**PUNCTUATION**	Exact match (`}`)

Total Tokens Count for Segment (3): 19

11 Find the Regular Expression corresponding to given statement, subset of {0,1}*
1. The Language of all strings containing at least one 0 and at least one
2. The Language of all strings containing 0's and 1's both are even.
3. The Language of all strings containing at most one pair of consecutive 1's.
4. The Language of all strings that do not end with 01.