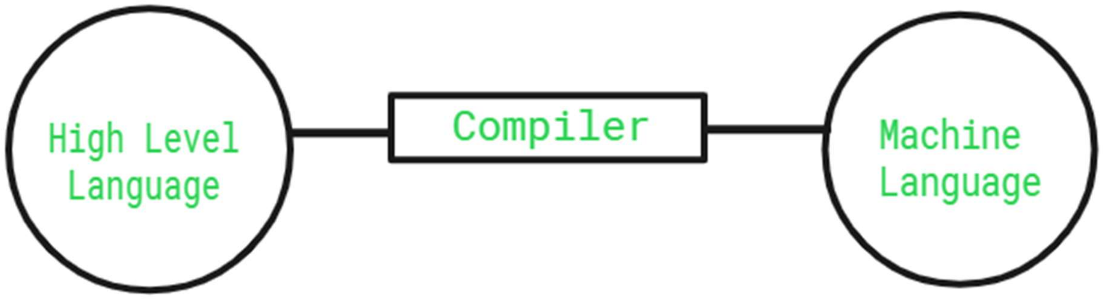
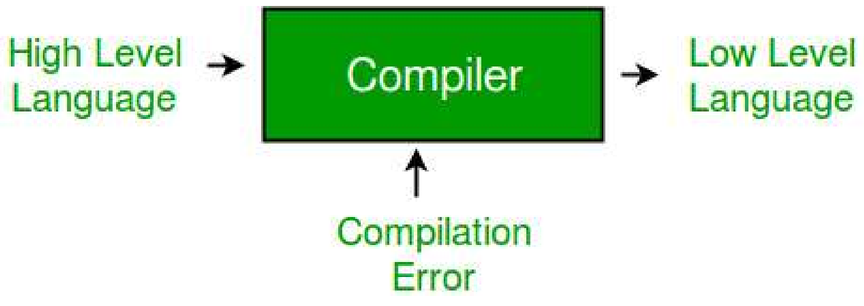
1.1 Introduction to Compiler

A compiler is a specialized software tool that translates a high-level programming language source code into machine code or an intermediate code. The primary purpose of a compiler is to facilitate the execution of a program on a computer by converting human-readable code written in a programming language into a form that can be understood and executed by the computer's hardware.





A compiler plays a crucial role in the software development process by translating human-readable source code into machine-executable code, enabling the execution of programs on a computer. The compilation process involves several stages, each addressing specific aspects of code analysis, optimization, and transformation.

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.

- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

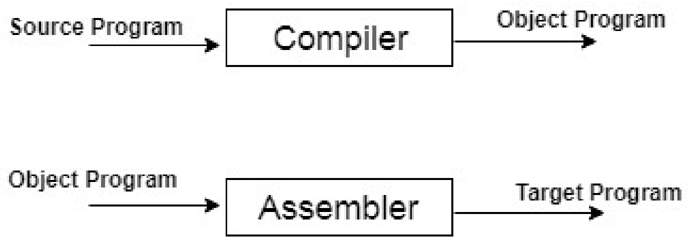
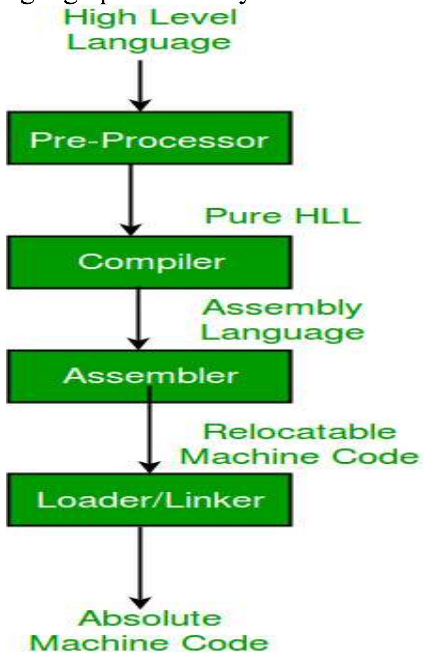


Fig: Execution process of source program in Compiler

Language Processing Systems

We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently, we tend to write programs in a high-level language, that is much less complicated for us to comprehend and maintain in our thoughts. Now, these programs go through a series of transformations so that they can readily be used by machines. This is where language procedure systems are used.



High-Level Language to Machine Code

High-Level Language: If a program contains pre-processor directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called preprocessor directives. They direct the pre-processor about what to do.

Pre-Processor: The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc.
Assembly Language: It's neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
Assembler: For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. Its translates assembly language to machine code.
Interpreter: An interpreter converts high-level language into low-level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing, and executes the source code whereas the interpreter does the same line by line. A compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower concerning compiled ones. For example: Let in the source program, it is written #include "Stdio. h". Pre-Processor replaces this file with its contents in the produced output. The basic work of a linker is to merge object codes (that have not even been connected), produced by the compiler, assembler, standard library function, and operating system resources. The codes generated by the compiler, assembler, and linker are generally re-located by their nature, which means to say, the starting location of these codes is not determined, which means they can be anywhere in the computer memory, Thus the basic task of loaders to find/calculate the exact address of these memory locations.
Relocatable Machine Code: It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
Loader/Linker: Loader/Linker converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

Compiler	**Assembler**	**Interpreter**
A compiler will take all the instructions written in a programming language and turns them into a secret language that the computer can understand and follow. This secret language is called machine code, and it's what makes the program run on the computer smoothly.	An Assembler is a special tool that translates programs written in Assembly language into machine code, which the computer can understand.	An Interpreter will run line by line and further translate each line accordingly after taking the source program
It needs a lot of memory for creating object codes.	Assembler will identify the error and then begins translating it into computer language	In this Processor, debugging is much easy since it translates until the error is found

The Compiler requires more time to analyse the entire source code however, the overall execution time is relatively faster.	Assembler is much faster than the compiler and equal to the speed of C	It needs less memory than the compiler
This Processor will outline the error after scanning the whole program, hence, debugging is hard since the error will be everywhere.	This Processor can save memory	No object code is generated.
It doesn't provide much security.	The Compiler requires more time to analyze the entire source code however, the overall execution time is relatively faster.	A compiler is more useful for the security purpose.
Examples: C, C++, C #	Examples: ADD, MUL, SUB, etc.	Examples: Python, Perl, JavaScript and Ruby

Assembler vs Compiler

Criteria	Compiler	Assembler
Definition	Compiler converts program written in a high-level language to machine-level language.	The assembler converts assembly code into machine code.
Debugging	Debugging is easy in the case of the compiler.	Debugging is tough as compared to the compiler.
Competency	The compiler is more intelligent than the assembler.	The assembler is less intelligent as compared to the compiler.
Phases	A compiler works in the following phases: lexical analyzer, semantic analyzer, syntax analyzer, intermediate code generator,	An assembler works in two phases over the given input: the first phase and the second phase.

	code optimizer, symbol table, and error handle.	
Conversion method	The compiler scans the entire program before converting it into machine code.	Assembler converts code into object code then it converts object code into machine code.
Examples	Examples of compilers are Clang, GCC, javac, etc.	Examples of assemblers are GNU, GAS, etc.

Compiler vs Interpreter

Criteria	**Compiler**	**Interpreter**
Translation	It translates a High-Level language into machine code at once.	It translates a High-Level language into machine code line by line.
Requirement of Translator Program	A translator Program is required each time for execution.	A translator Program is not required for execution.
Object File Creation	It creates and stores an object file.	It does not create an object program.
Memory	Memory consumption is more in the case of the compiler.	An interpreter takes relatively less memory than the compiler.
Debugging	Debugging of code is relatively harder.	Debugging is easy, compared to the compiler.
Cost	A compiler is relatively costlier than an interpreter.	Less costly.
Security	More useful in the case of security.	Interpreter is more vulnerable to threats.
Execution time	Execution time is less.	Execution time is higher.

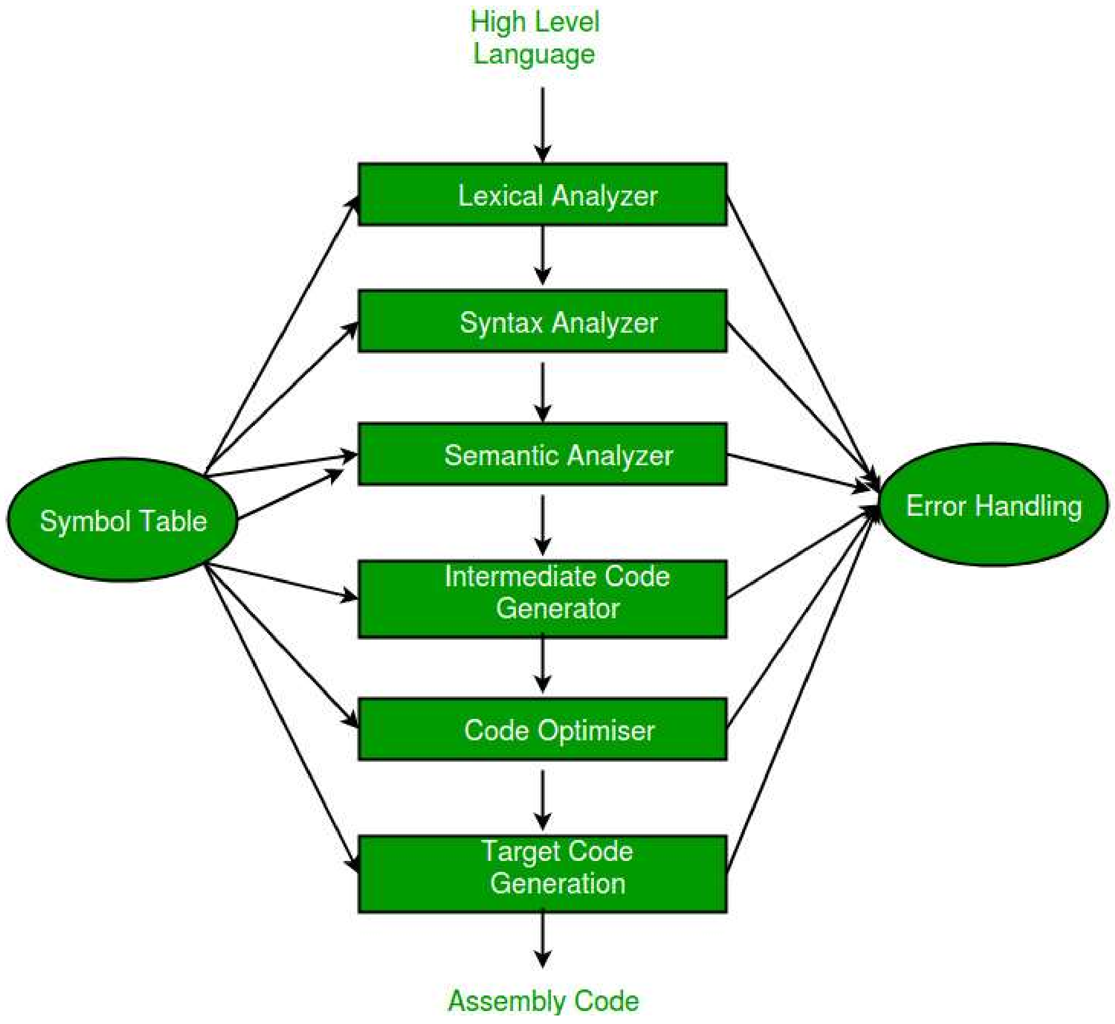
Suitability	Suitable for large programs.	Suitable for small programs.

1.2 Phases and Passes

In the context of a compiler, the terms "phases" and "passes" refer to different stages of the compilation process, each with a specific set of tasks and responsibilities. These terms are often used interchangeably, but they have distinct meanings.

1. Phases:

- A compiler is typically organized into distinct phases, where each phase corresponds to a high-level task in the compilation process. These phases represent a logical division of the overall compilation process, helping in the systematic analysis and transformation of the source code. The most common phases in a compiler include:
- Lexical Analysis: Identifying and tokenizing the source code.
- Syntax Analysis: Building a syntax tree to represent the grammatical structure of the code.
- Semantic Analysis: Checking for semantic errors and building a symbol table.
- Intermediate Code Generation: Producing an intermediate code representation.
- Code Optimization: Improving the efficiency of the intermediate code.
- Code Generation: Translating the optimized code into machine code or assembly language.
- Code Linking and Assembly: Combining and linking different modules to create an executable.



We basically have two phases of compilers, namely the Analysis phase and Synthesis phase. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation.

The analysis of a source program is divided into mainly three phases. They are:
Linear Analysis-
This involves a scanning phase where the stream of characters is read from left to right. It is then grouped into various tokens having a collective meaning.
Hierarchical Analysis-
In this analysis phase, based on a collective meaning, the tokens are categorized hierarchically into nested groups.
Semantic Analysis-
This phase is used to check whether the components of the source program are meaningful or not.
The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

Here's a breakdown of the key components and processes involved in the functioning of a compiler:

1. Lexical Analysis:

- The first phase of a compiler is lexical analysis, where the source code is broken down into a sequence of tokens. Tokens are the smallest units in a programming language, such as keywords, identifiers, literals, and operators.

- Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High -level input program into a sequence of **Tokens**.
- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

What is a token?

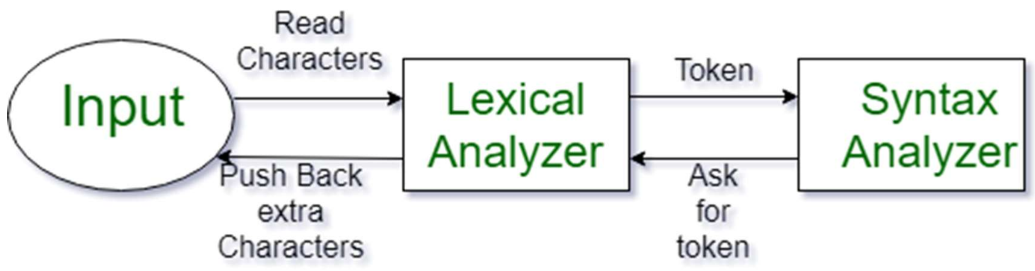
- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)
- Keywords; Examples-for, while, if etc.
- Identifier; Examples-Variable name, function name, etc.
- Operators; Examples '+', '++', '-' etc.
- Separators; Examples ',' ';' etc

Example of Non-Tokens:

- Comments, preprocessor directive, macros, blanks, tabs, newline, etc.



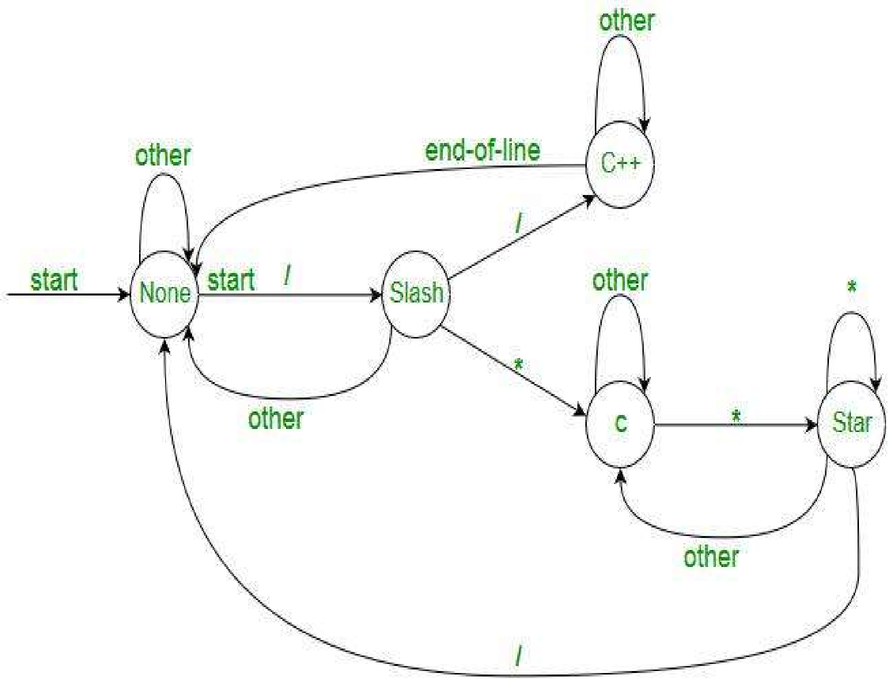
Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

How Lexical Analyzer works-

— Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.

— Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

- Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
- Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
- Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.



The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.
Suppose we pass a statement through lexical analyzer – **a = b + c;** It will generate token sequence like this: **id=id+id**; Where each id refers to it's variable in the symbol table referencing all details For example, consider the program

```
int main()
{
 // 2 variables
 int a, b;
 a = 10;
return 0;
}
```

All the valid tokens are:
'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';'
'a' '=' '10' ';' 'return' '0' ';' '}'
Above are the valid tokens. You can observe that we have omitted comments.

Exercise 1: Count number of tokens:

```
int main()
{
 int a = 10, b = 20;
 printf("sum is:%d",a+b);
 return 0;
}
```

Answer: Total number of tokens: 27.

Exercise 2: Count number of tokens: int max(int i);

Lexical analyzer first read int and finds it to be valid and accepts as token.
max is read by it and found to be a valid function name after reading (
int is also a token , then again I as another token and finally ;
Answer: Total number of tokens 7:
int, max, (,int, i,), ;
We can represent in the form of lexemes and tokens as under:

Lexemes	Tokens	Lexemes	Tokens
while	WHILE	a	IDENTIEFIER
(	LAPREN	=	ASSIGNMENT
a	IDENTIFIER	a	IDENTIFIER
>=	COMPARISON	–	ARITHMETIC
b	IDENTIFIER	2	INTEGER
)	RPAREN	;	SEMICOLON

2. Syntax Analysis:
- In this phase, the compiler analyses the syntactic structure of the source code using a formal grammar. It creates a hierarchical structure known as the Abstract Syntax Tree (AST), representing the syntactic relationships between different components of the code.

- When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.

- Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or - **Syntax tree.** The parse tree is constructed by using the pre-

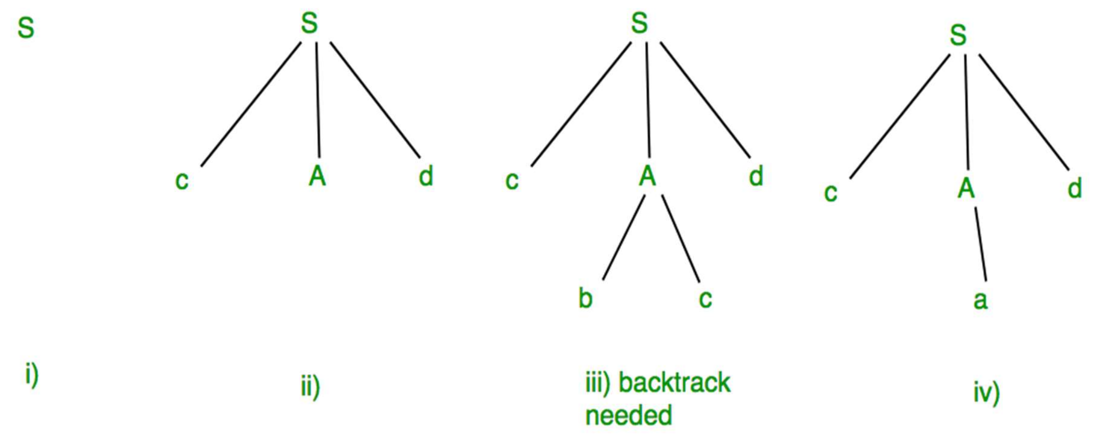
defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.
- Syntax analysis, also known as parsing, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language. This is typically the second stage of the compilation process, following lexical analysis.
- The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.
- There are several types of parsing algorithms used in syntax analysis, including:
LL parsing: This is a top-down parsing algorithm that starts with the root of the parse tree and constructs the tree by successively expanding non-terminals. LL parsing is known for its simplicity and ease of implementation.
LR parsing: This is a bottom-up parsing algorithm that starts with the leaves of the parse tree and constructs the tree by successively reducing terminals. LR parsing is more powerful than LL parsing and can handle a larger class of grammars.
LR(1) parsing: This is a variant of LR parsing that uses lookahead to disambiguate the grammar.
LALR parsing: This is a variant of LR parsing that uses a reduced set of lookahead symbols to reduce the number of states in the LR parser.
- Once the parse tree is constructed, the compiler can perform semantic analysis to check if the source code makes sense and follows the semantics of the programming language.
- The parse tree or AST can also be used in the code generation phase of the compiler design to generate intermediate code or machine code.

Features of syntax analysis:
Syntax Trees: Syntax analysis creates a syntax tree, which is a hierarchical representation of the code's structure. The tree shows the relationship between the various parts of the code, including statements, expressions, and operators.
Context-Free Grammar: Syntax analysis uses context-free grammar to define the syntax of the programming language. Context-free grammar is a formal language used to describe the structure of programming languages.
Top-Down and Bottom-Up Parsing: Syntax analysis can be performed using two main approaches: top-down parsing and bottom-up parsing. Top-down parsing starts from the highest level of the syntax tree and works its way down, while bottom-up parsing starts from the lowest level and works its way up.
Error Detection: Syntax analysis is responsible for detecting syntax errors in the code. If the code does not conform to the rules of the programming language, the parser will report an error and halt the compilation process.
Intermediate Code Generation: Syntax analysis generates an intermediate representation of the code, which is used by the subsequent phases of the compiler. The intermediate representation is usually a more abstract form of the code, which is easier to work with than the original source code.
Optimization: Syntax analysis can perform basic optimizations on the code, such as removing redundant code and simplifying expressions.
- The pushdown automata (PDA) is used to design the syntax analysis phase.
- The Grammar for a Language consists of Production rules.

Example: Suppose Production rules for the Grammar of a language are:
S -> cAd

A -> bc|a
And the input string is "cad".

- Now the parser attempts to construct a syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string "cad" it uses the rules as shown in the given diagram:



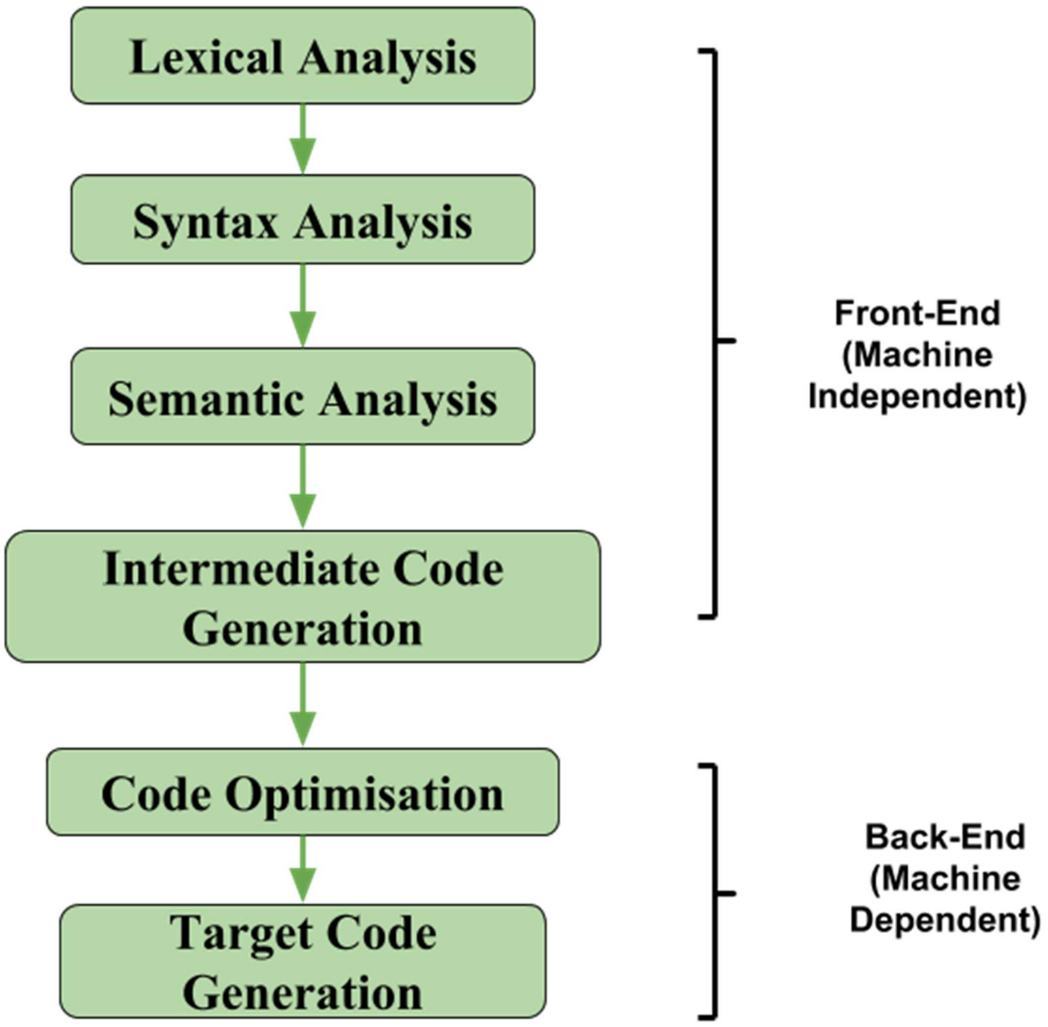
- In step (iii) above, the production rule A->bc was not a suitable one to apply (because the string produced is "cbcd" not "cad"), here the parser needs to backtrack, and apply the next production rule available with A which is shown in step (iv), and the string "cad" is produced.
- Thus, the given input can be produced by the given grammar, therefore the input is correct in syntax. - But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.
There can be an easier way to solve this, which we shall study in next coming units under FIRST and FOLLOW topic.

3. Semantic Analysis:
- The compiler checks for semantic errors and ensures that the program adheres to the rules and constraints of the programming language. It also performs type checking and resolves references to variables and functions.

4. Intermediate Code Generation:
- The compiler translates the source code into an intermediate code, which is a platform-independent and language-agnostic representation of the program. This intermediate code serves as an intermediate step between the source code and the target machine code.

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent intermediate code are:
-Because of the machine-independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
-Retargeting is facilitated.

-It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

-If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language. independent (three-address code). The following are commonly used intermediate code representations:

Postfix Notation:

- Also known as reverse Polish notation or suffix notation.
- In the infix notation, the operator is placed between operands, e.g., a + b. Postfix notation positions the operator at the right end, as in ab +.
- For any postfix expressions e1 and e2 with a binary operator (+) , applying the operator yields e1e2+.
- Postfix notation eliminates the need for parentheses, as the operator’s position and arity allow unambiguous expression decoding.

- In postfix notation, the operator consistently follows the operand.
  Example 1: The postfix representation of the expression (a + b) * c is : ab + c *
  Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

Three-Address Code:

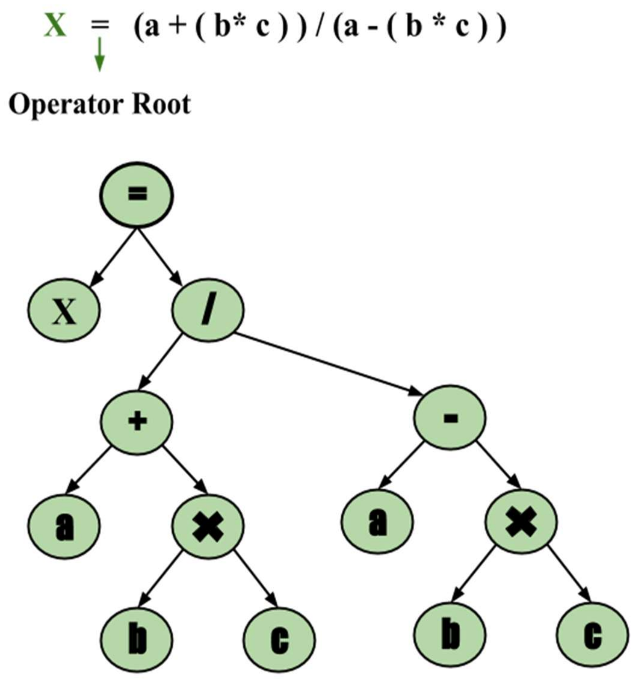
- A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
- A sequence of three address statements collectively forms a three address code.
- The typical form of a three address statement is expressed as x = y op z, where x, y, and z represent memory addresses.
- Each variable (x, y, z) in a three address statement is associated with a specific memory location.
- While a standard three address statement includes three references, there are instances where a statement may contain fewer than three references, yet it is still categorized as a three address statement.
  Example: The three address code for the expression a + b * c + d : T1 = b * c T2 = a + T1 T3 = T2 + d; T 1 , T2 , T3 are temporary variables.

  There are 3 ways to represent a Three-Address Code in compiler design:
  i) Quadruples
  ii) Triples
  iii) Indirect Triples

Syntax Tree:

- A syntax tree serves as a condensed representation of a parse tree.
- The operator and keyword nodes present in the parse tree undergo a relocation process to become part of their respective parent nodes in the syntax tree. the internal nodes are operators and child nodes are operands.
- Creating a syntax tree involves strategically placing parentheses within the expression. This technique contributes to a more intuitive representation, making it easier to discern the sequence in which operands should be processed.
- The syntax tree not only condenses the parse tree but also offers an improved visual representation of the program's syntactic structure,

- Example: x = (a + b * c) / (a – b * c)



5. Code Optimization:

- During this phase, the compiler performs various optimizations to improve the efficiency and performance of the generated code. This may include techniques such as constant folding, loop optimization, and dead code elimination.

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following **Objectives:**

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize?

Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Why Optimize?

- Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:
- Reduce the space consumed and increases the speed of compilation.

- Manually analyzing datasets involves a lot of time. Hence, we make use of software like Tableau for data analysis. Similarly, manually performing the optimization is also tedious and is better done using a code optimizer.
- An optimized code often promotes re-usability.

Types of Code Optimization: The optimization process can be broadly classified into two types:

- **Machine Independent Optimization:** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
- **Machine Dependent Optimization:** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

6. Target Code Generator – The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

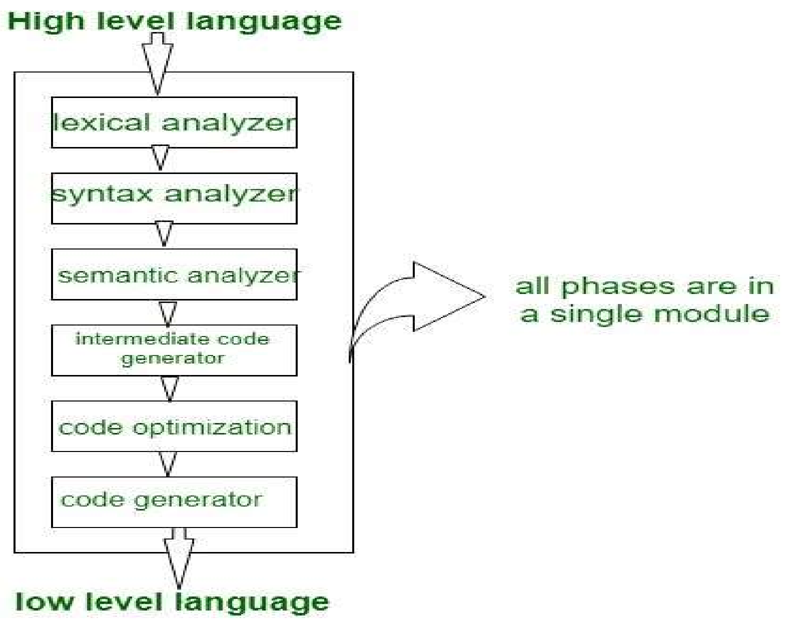
7. Error Handling:
- Throughout the compilation process, the compiler detects and reports various errors, such as syntax errors, semantic errors, or inconsistencies in the source code. It provides meaningful error messages to help programmers identify and correct issues in their code.

8. Symbol Table:
- The symbol table is defined as the set of Name and Value pairs.
- Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc. and is managed by Symbol Table Manager.
- **Symbol Table –** It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

2. Passes:
A **Compiler pass** refers to the traversal of a compiler through the entire program. Compiler passes are of two types Single Pass Compiler, and Two Pass Compiler *or* Multi-Pass Compiler. These are explained as follows.

Types of Compiler Pass

1. Single Pass Compiler
If we combine or group all the phases of compiler design in a **single** module known as a single pass compiler.



Single Pass Compiler

In the above diagram, there are all 6 phases are grouped in a single module, some points of the single pass compiler are as:

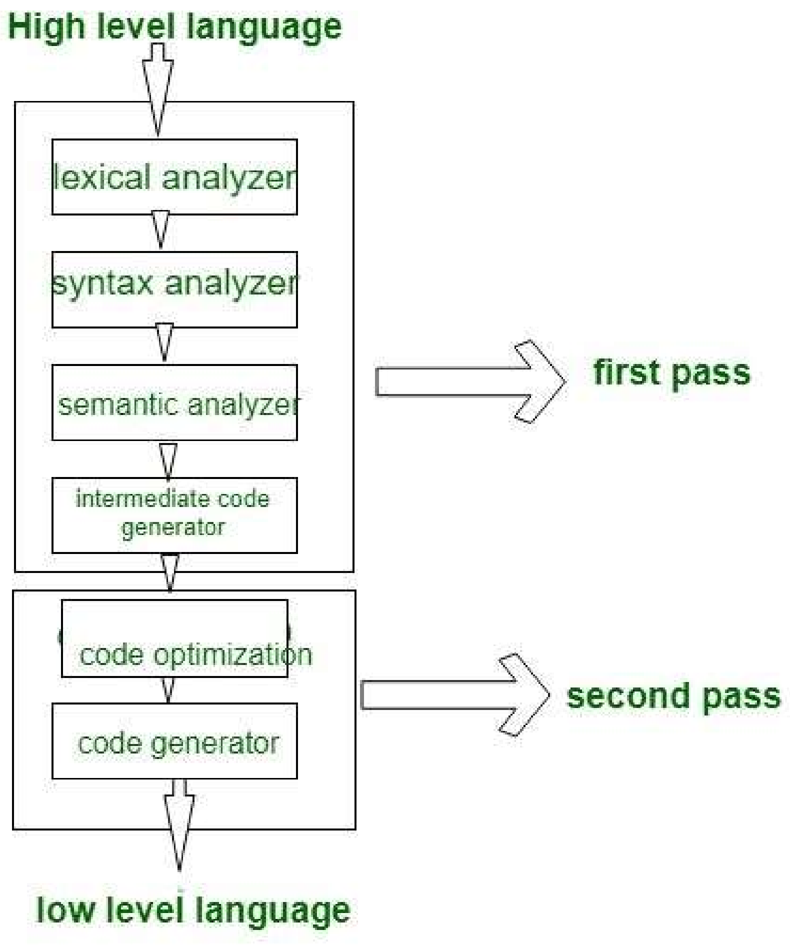
- A one-pass/single-pass compiler is a type of compiler that passes through the part of each compilation unit exactly once.
- Single pass compiler is faster and smaller than the multi-pass compiler.
- A disadvantage of a single-pass compiler is that it is less efficient in comparison with the multipass compiler.
- A single pass compiler is one that processes the input *exactly once*, so going directly from lexical analysis to code generator, and then going back for the next read.

Problems with Single Pass Compiler

- We cannot optimize very well due to the context of expressions are limited.
- As we can't back up and process it again so grammar should be limited or simplified.
- Command interpreters such as *bash/sh/tcsh* can be considered Single pass compilers, but they also execute entries as soon as they are processed.

2. Two-Pass compiler or Multi-Pass compiler

A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. In multi-pass Compiler, we divide phases into two passes as:



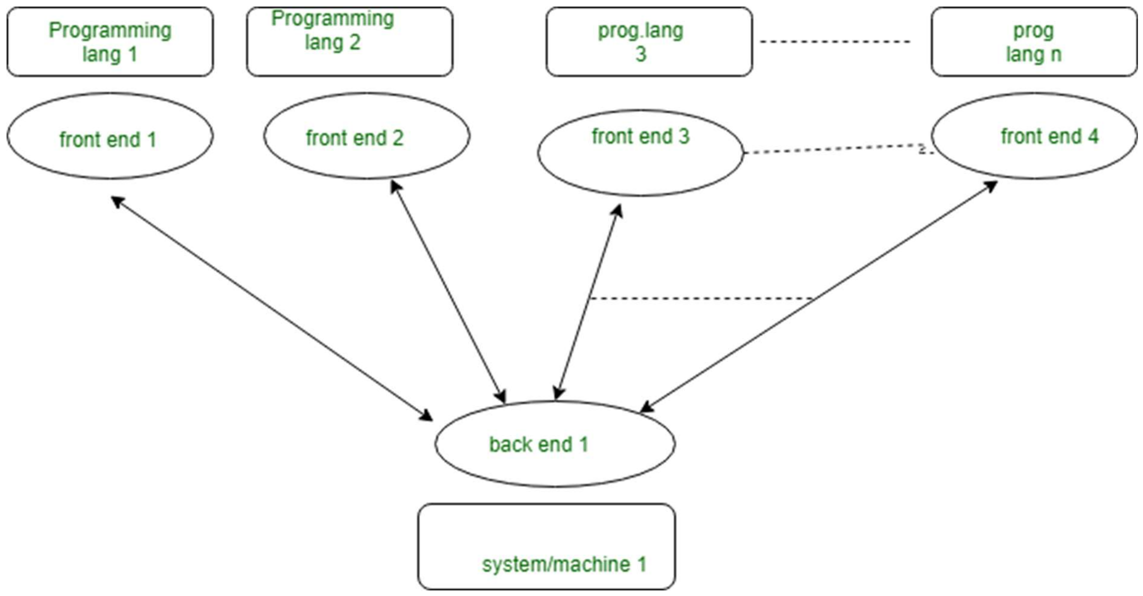
First Pass is referred as

- Front end
- Analytic part
- Platform independent

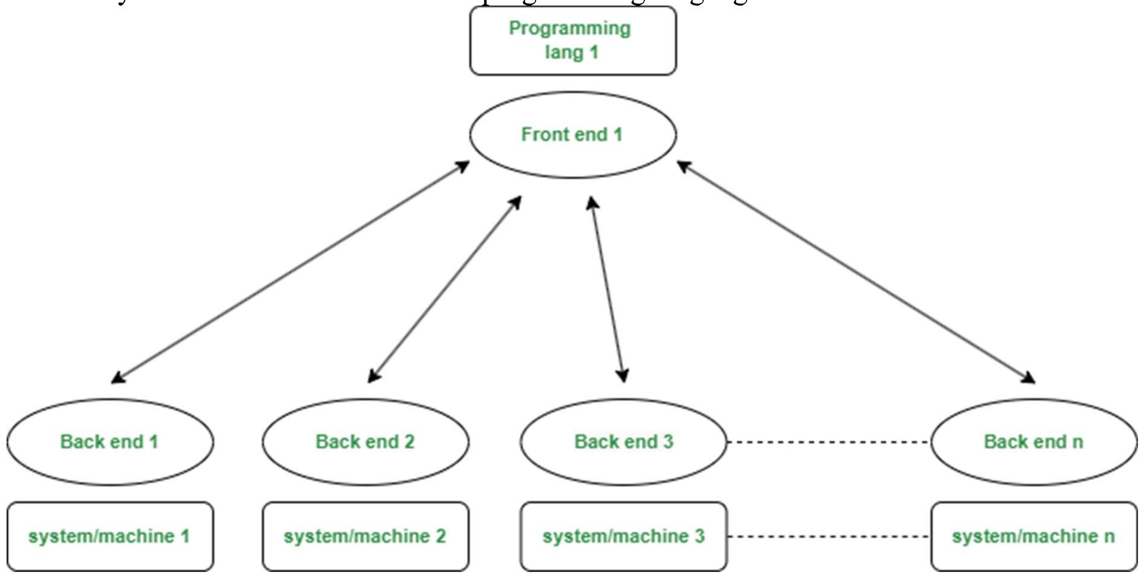
Second Pass is referred as

- Back end
- Synthesis Part
- Platform Dependent
-

Problems that can be Solved With Multi-Pass Compiler
First: If we want to design a compiler for a different programming language for the same machine. In this case for each programming language, there is a requirement to make the Front end/first pass for each of them and only one Back end/second pass as:



Second: If we want to design a compiler for the same programming language for different machines/systems. In this case, we make different Back end for different Machine/system and make only one **Front end** for the same programming language as:



1.3 Bootstrapping

Bootstrapping refers to the process of using a compiler to compile its own source code. In other words, a compiler is initially built with the help of another pre-existing compiler (often referred to as the "host compiler" or "bootstrap compiler"), and once it is successfully constructed, it can then be used to compile its own source code. This self-compilation process is known as bootstrapping.
The bootstrapping process typically involves the following steps:

1. Initial Compiler (Bootstrap Compiler):

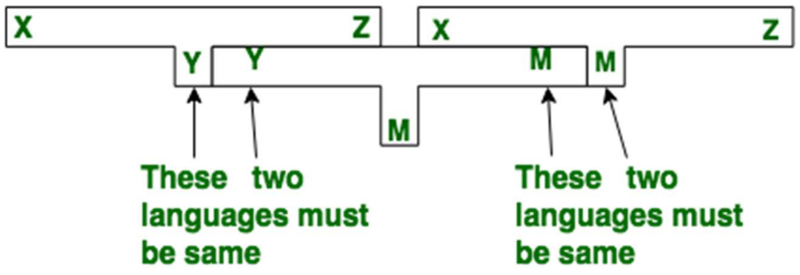
- A compiler for a programming language is initially created using another existing compiler. This compiler is often written in a different language and is referred to as the bootstrap compiler.

2. Compiling the Compiler Source Code:
- The bootstrap compiler is then used to compile the source code of the compiler written in the target language. This results in a new version of the compiler.

3. Replacing the Bootstrap Compiler:
- The newly compiled compiler is now capable of compiling its own source code. The bootstrap compiler is replaced with this self-compiled version.

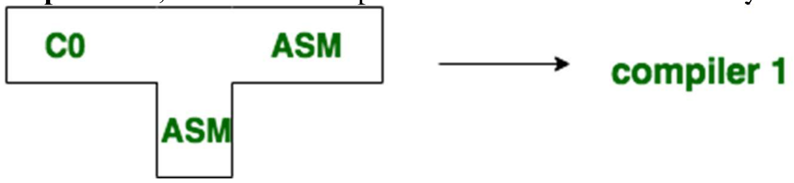
4. Iterative Process:
- The process can be repeated iteratively. With each iteration, the compiler is improved or modified, and the updated version is compiled using the previous version. This cycle continues, leading to the development of more advanced compiler versions.

- Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.
- Writing a compiler for any high-level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.
- To clearly understand the Bootstrapping technique, consider a following scenario. Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is saying Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.
- Following diagram illustrates the above scenario. Example: We can create compiler of many different forms.

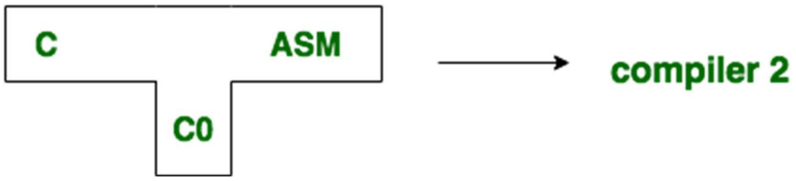

X
Z
Y
Y
M
X
Z
M
M
These two languages must be same
These two languages must be same

- Now we will generate a Compiler which takes C language and generates an assembly language as an output with the availability of a machine of assembly language.

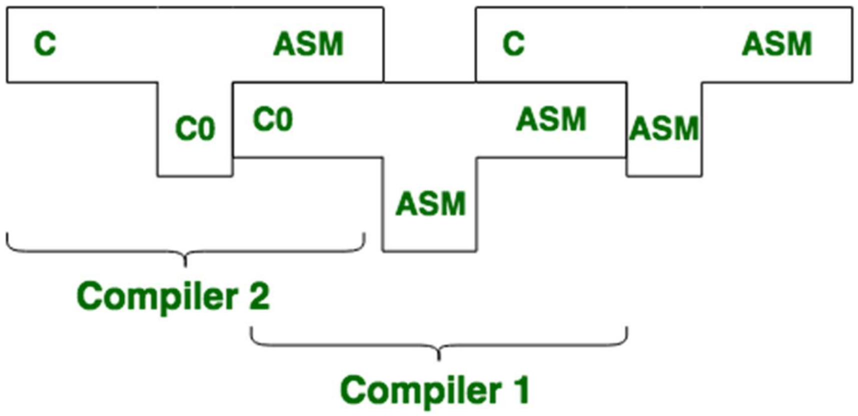
Step-1: First, we write a compiler for a small of C in assembly language.



Step-2: Then using with small subset of C i.e. C0, for the source language c the compiler is written.



Step-3: Finally, we compile the second compiler. using compiler 1 the compiler 2 is compiled.



Step-4: Thus, we get a compiler written in ASM which compiles C and generates code in ASM.

- Bootstrapping is the process of writing a compiler for a programming language using the language itself. In other words, it is the process of using a compiler written in a particular programming language to compile a new version of the compiler written in the same language.
- The process of bootstrapping typically involves several stages. In the first stage, a minimal version of the compiler is written in a different language, such as assembly language or C. This minimal version of the compiler is then used to compile a slightly more complex version of the compiler written in the target language. This process is repeated until a fully functional version of the compiler is written in the target language.
- There are several advantages to bootstrapping.
  - One advantage is that it ensures that the compiler is compatible with the language it is designed to compile. This is because the compiler is written in the same language, so it is better able to understand and interpret the syntax and semantics of the language.
  - Another advantage is that it allows for greater control over the optimization and code generation process. Since the compiler is written in the target language, it can be optimized to generate code that is more efficient and better suited to the target platform.

- However, bootstrapping also has some disadvantages.
    - One disadvantage is that it can be a time-consuming process, especially for complex languages or compilers.
    - It can also be more difficult to debug a bootstrapped compiler, since any errors or bugs in the compiler will affect the subsequent versions of the compiler.

Bootstrapping is essential for several reasons:

- **Independence:** Once the bootstrapping process is complete, the compiler becomes independent of the initial bootstrap compiler. It can be used to generate executable code for programs written in the target language without relying on the bootstrap compiler.

- **Compiler Development:** Bootstrapping facilitates the development of compilers for new programming languages or the enhancement of existing ones. It allows for the creation of compilers in a self-sustaining manner.

- **Portability:** Bootstrapping ensures that the compiler can run on different platforms and architectures. As long as there is a bootstrap compiler available for a specific platform, the compiler can be generated for that platform.

- **Trustworthiness:** The bootstrapping process helps establish trust in the correctness of the compiler. If the compiler can successfully compile its own source code, it provides confidence in its ability to correctly translate source code written in the target language.

1.4 Finite State Machines and Regular Expressions and their Applications to Lexical Analysis

Finite State Machines (FSMs) and Regular Expressions are fundamental concepts in computer science, particularly in the field of formal language theory and lexical analysis.

1. Finite State Machines (FSMs):

A Finite State Machine is a mathematical model used to describe the behavior of a system that can be in a finite number of states. It consists of states, transitions between states, and an initial state. FSMs are widely used for modeling and analyzing the behavior of systems with discrete and sequential logic.

In the context of lexical analysis in compiler design, FSMs are employed to recognize and process tokens in the input source code. The lexical analysis phase involves scanning the source code to identify and categorize the basic building blocks, known as tokens. Each token is associated with a specific pattern or regular language.
2. Regular Expressions:

Regular Expressions (regex or regexp) are a powerful way to represent patterns in strings. They are used to describe sets of strings by defining a pattern of characters. Regular expressions can be simple, such as matching a specific word, or complex, specifying more intricate patterns. They are widely used in text processing, searching, and lexical analysis.

Applications to Lexical Analysis:

In the context of compiler design, FSMs and regular expressions work together in the lexical analysis phase to recognize and categorize tokens in the source code. Here's how they are applied:

a. Token Recognition with Regular Expressions:
- Regular expressions are used to define the patterns of different tokens in the programming language. For example, a regular expression might define the pattern for identifying keywords, identifiers, numeric literals, or operators.

b. FSMs for Tokenization:
- FSMs are employed to implement the recognition of tokens based on the regular expressions. Each state in the FSM corresponds to a specific phase of token recognition. Transitions between states are determined by the characters in the input stream.

c. Combining FSMs and Regular Expressions:
- The combination of FSMs and regular expressions allows for an efficient and modular approach to lexical analysis. Each FSM corresponds to a particular type of token, and regular expressions define the patterns that trigger state transitions.

d. Lexical Analyzer Generator Tools:
- Lexical analyser generator tools, such as Lex or Flex, use regular expressions to define the token patterns and automatically generate the corresponding FSM-based lexical analysers. These tools simplify the process of creating the lexical analysis component of a compiler.

e. Performance and Efficiency:
- FSMs, being deterministic in nature, contribute to the efficiency of lexical analysis. The use of regular expressions allows concise and expressive definitions of token patterns.

In summary, FSMs and regular expressions are essential tools in the lexical analysis phase of compiler construction. Regular expressions define the patterns of tokens, while FSMs, often implemented as state transition diagrams or tables, are used to recognize and process these patterns efficiently. The combination of these concepts forms the foundation for building robust and efficient lexical analyzers in compiler design.

1.4.1 Finite State Machines and its Applications to Lexical Analysis

Finite State Machines (FSMs) play a crucial role in lexical analysis, which is the first phase of a compiler responsible for scanning the source code and identifying tokens. Here's an explanation of Finite State Machines and their applications to lexical analysis:

1. Finite State Machines (FSMs):

A Finite State Machine is a mathematical model used to represent systems with a finite number of states and transitions between these states. FSMs are widely applied in various fields to describe the behaviour of systems with discrete and sequential logic. In the context of lexical analysis, FSMs are used to recognize patterns in the input source code and identify tokens.

2. Applications of FSMs to Lexical Analysis:

a. Token Recognition:
- In lexical analysis, the source code is scanned character by character to identify tokens, which are the basic building blocks of a programming language (e.g., keywords, identifiers, literals, operators). FSMs are used to recognize these tokens based on patterns defined by regular expressions.

b. Lexical Specification:
- The lexical structure of a programming language is often specified using regular expressions, which describe the patterns of valid tokens. FSMs are employed to implement the recognition of these patterns. Each FSM corresponds to a particular token type.

c. Deterministic Finite Automaton (DFA):
- A DFA is a specific type of FSM where each transition is uniquely determined by the current state and the input symbol. Lexical analysers often use DFAs for efficiency. The states in the DFA represent different stages in the recognition of a token, and transitions between states are based on the characters in the input stream.

d. State Transition Diagrams:
- FSMs in lexical analysis are often represented using state transition diagrams. These diagrams visually illustrate the states, transitions, and accepting states of the FSM. Each state corresponds to a specific stage in token recognition, and transitions occur based on the input characters.

e. Lexical Analyzer Design:
- Lexical analysers, also known as lexers or scanners, are components of compilers responsible for tokenizing the source code. FSMs are employed in the design and implementation of lexical analysers to efficiently recognize and categorize tokens.

f. Lexical Analyzer Generator Tools:
- Lexical analyser generator tools, such as Lex or Flex, use FSMs to implement the token recognition process. These tools take as input a set of regular expressions that define the token patterns and automatically generate the corresponding FSM-based lexical analysers.

g. Efficiency and Performance:
- FSMs contribute to the efficiency of lexical analysis. Due to their deterministic nature, FSM-based lexical analyzers can quickly process the input stream and recognize tokens without backtracking.

In summary, FSMs are a fundamental concept in lexical analysis, providing a systematic and efficient way to recognize tokens based on patterns specified by regular expressions. The use of FSMs contributes to the design of robust and high-performance lexical analyzers in the compilation process.

1.4.2 Regular Expressions and their Applications to Lexical Analysis

Regular expressions (regex or regexp) are powerful and concise notations used to describe patterns in strings. In the context of lexical analysis, regular expressions play a crucial role in specifying the patterns of tokens in a programming language. Here's an explanation of regular expressions and their applications to lexical analysis:

1. Regular Expressions:

Regular expressions are patterns that define sets of strings. They consist of a combination of characters and special symbols that represent matching rules. The basic building blocks of regular expressions include:

- **Literal Characters:** Represent themselves.
- **Metacharacters:** Special characters with a specific meaning, such as '.', '*', '+', '?', etc.
- **Character Classes:** Specify a set of characters, e.g., `[0-9]` represents any digit.
- **Quantifiers:** Indicate the number of occurrences, e.g., '*', '+', '?'.
- **Grouping:** Parentheses are used to group elements.
- **Anchors:** '^' (caret) and '$' (dollar sign) represent the beginning and end of a line, respectively.

2. Applications of Regular Expressions to Lexical Analysis:

a. Token Specification:
- Regular expressions are used to specify the patterns of different tokens in a programming language. Tokens include keywords, identifiers, literals (e.g., numbers and strings), operators, and other syntactic elements.

b. Lexical Structure Definition:
- The lexical structure of a programming language is often defined using regular expressions. For example, the regular expression for an identifier might be `[a-zA-Z_][a-zA-Z0-9_]*`, representing a letter or underscore followed by zero or more letters, digits, or underscores.

c. Tokenization:
- During lexical analysis, the source code is scanned character by character. Regular expressions are employed to recognize and tokenize the input based on the defined patterns. For example, a regular expression might describe the pattern of a numeric literal or a keyword.

d. Lexical Analyzer Generator Tools:
- Lexical analyzer generator tools, such as Lex or Flex, use regular expressions to specify the lexical rules of a programming language. Programmers provide regular expressions for different token types, and the tool generates a lexical analyzer (scanner) based on these specifications.

e. Pattern Matching:
- Regular expressions are used for pattern matching in the input stream. Lexical analyzers apply regular expressions to recognize patterns and categorize portions of the input as specific tokens.

f. Conciseness and Expressiveness:
- Regular expressions offer a concise and expressive way to represent complex patterns. This makes it easier for programmers and language designers to define the lexical structure of a language in a clear and compact manner.

g. Flexibility in Token Definition:
- Regular expressions provide a flexible and extensible way to define tokens. Changes or additions to the language's lexical rules can be easily accommodated by modifying or adding regular expressions.

In summary, regular expressions are a fundamental tool in lexical analysis, providing a compact and expressive means to specify the patterns of tokens in a programming language. They are widely used in the design and implementation of lexical analyzers, contributing to the efficiency and flexibility of the compilation process.

1.5 Implementation of Lexical Analysers

The implementation of lexical analyzers involves writing code that scans the input source code and identifies tokens based on specified lexical rules. Here, I'll provide a high-level overview of the steps involved in implementing a lexical analyzer, and I'll use a hypothetical language for illustration purposes.

Let's consider a simple language with the following token types: identifiers, keywords, numeric literals, and operators.

```python
# Sample input code
input_code = "int main() { return 0; }"
```

1. Define Lexical Rules using Regular Expressions:

Specify regular expressions for each token type based on the language's syntax. For example:

```python
import re

token_patterns = [
    ('IDENTIFIER', r'[a-zA-Z_][a-zA-Z0-9_]*'),
    ('KEYWORD', r'int|return'),
    ('NUMERIC_LITERAL', r'\d+'),
    ('OPERATOR', r'\+|\-|\*|\/|\=|\{|\}'),
    ('WHITESPACE', r'\s+'),
]

# Combine regular expressions into a single pattern
combined_pattern = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in token_patterns)

# Compile the regular expression pattern
lexer_pattern = re.compile(combined_pattern)
```

2. Implement Lexical Analyzer:

Write code for the lexical analyzer that iterates through the input code, matches regular expressions, and identifies tokens:

```python
def lexical_analyzer(input_code):
    tokens = []
    position = 0

    while position < len(input_code):
        match = lexer_pattern.match(input_code, position)

        if match:
            for name, value in match.groupdict().items():
                if value:
                    tokens.append((name, value))
            position = match.end()
        else:
            raise Exception(f"Lexical error at position {position}: Unexpected character
'{input_code[position]}'")

    return tokens
```

3. Test the Lexical Analyzer:

Test the lexical analyzer with sample input code:

```python
tokens = lexical_analyzer(input_code)
print(tokens)
```

The output would be a list of identified tokens:

```python
[('KEYWORD', 'int'), ('IDENTIFIER', 'main'), ('OPERATOR', '('), ('OPERATOR', ')'),
('OPERATOR', '{'), ('KEYWORD', 'return'), ('NUMERIC_LITERAL', '0'), ('OPERATOR',
';'), ('OPERATOR', '}')]
```

This is a simple example, and in a real-world scenario, you would need to handle more complex language features, error handling, and possibly incorporate more advanced techniques like using state machines.

In practice, tools like Lex, Flex, or other lexer generator tools are commonly used to generate lexical analyzers from high-level specifications, making the implementation more automated and efficient.

1.6 Lexical analyser Generator

A Lexical Analyzer Generator is a tool that automates the process of generating lexical analyzers (also known as lexers or scanners) based on a set of user-defined specifications. These tools simplify the implementation of the lexical analysis phase in compiler construction. Lexical analyzer generators typically take high-level specifications, often written in the form of regular expressions or patterns, and generate code for the corresponding lexical analyzers.

One of the most well-known lexical analyzer generator tools is Lex. Lex is often used in combination with the yacc (Yet Another Compiler Compiler) parser generator to create complete compiler front ends.

Here's a brief overview of how a lexical analyzer generator like Lex works:

1. Specification of Token Patterns:
- Users provide a set of regular expressions or patterns that define the syntax of different tokens in the programming language.

2. Actions for Token Recognition:
- Along with each regular expression, users can specify actions or code snippets to be executed when a particular pattern is recognized. These actions typically involve processing the matched text or returning information about the recognized token.

3. Generation of Lexical Analyzer Code:
- The lexical analyzer generator takes the user-defined specifications and generates source code for the lexical analyzer. The generated code is often written in a programming language like C or C++.

4. Integration with Compiler Front End:
- The generated lexical analyzer code is integrated into the overall compiler front end along with other components like parsers and semantic analyzers.

5. Compilation of Generated Code:
- The generated code is compiled to create an executable program, which is then used to scan and tokenize the input source code during the compilation process.

One of the widely used Lex implementations is Flex (Fast Lexical Analyzer Generator), which is an enhanced and more flexible version of Lex. Flex is compatible with Lex but provides additional features and optimizations.

Here's a simple example of how Lex/Flex specifications look:

~~~
```lex
%{
#include <stdio.h>
%}

%%

int            printf("Integer\n");
float          printf("Float\n");
[a-zA-Z_]+      printf("Identifier\n");
[0-9]+\.[0-9]+  printf("Float Literal\n");
[0-9]+          printf("Integer Literal\n");
.              printf("Unrecognized Character\n");

%%

int main() {
    yylex();
    return 0;
}
```
~~~

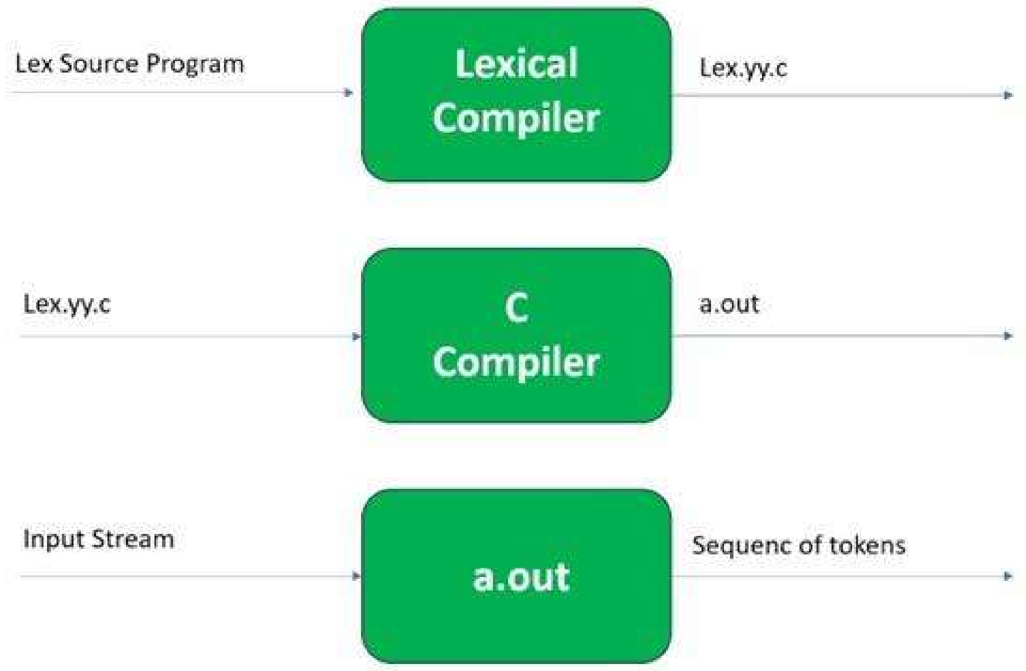
In this example, token patterns and corresponding actions are defined using regular expressions. The Lex/Flex tool generates C code for the lexical analyzer based on these specifications.

Using Lexical Analyzer Generators like Lex or Flex simplifies the implementation of the lexical analysis phase, making it more modular, maintainable, and adaptable to changes in the language's syntax.

1.7 LEX-compiler

Lexical Analysis
It is the first step of compiler design, it takes the input as a stream of characters and gives the output as tokens also known as tokenization. The tokens can be classified into identifiers, Sperators, Keywords, Operators, Constant and Special Characters.
It has three phases:
Tokenization: It takes the stream of characters and converts it into tokens.
Error Messages: It gives errors related to lexical analysis such as exceeding length, unmatched string, etc.
Eliminate Comments: Eliminates all the spaces, blank spaces, new lines, and indentations.
Lex
Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler. The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC(Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.
Function of Lex

1. In the first step the source code which is in the Lex language having the file name ‘File.l’ gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.
2. After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an ‘a.out’ file, and finally, the output file a.out will take the stream of character and generates tokens as output.
lex.yy.c: It is a C program.
File.l: It is a Lex source program
a.out: It is a Lexical analyzer



Block Diagram of Lex

Lex File Format

A Lex program consists of three parts and is separated by %% delimiters:-

Declarations
%%
Translation rules
%%
Auxiliary procedures

Declarations: The declarations include declarations of variables.
Transition rules: These rules consist of Pattern and Action.
Auxiliary procedures: The Auxilary section holds auxiliary functions used in the actions.
For example:
declaration
number[0-9]

LEX is not a compiler itself; rather, it is a tool for generating lexical analysers or scanners. LEX is commonly used in combination with YACC (Yet Another Compiler Compiler) to build a complete compiler.

Here's a brief explanation of LEX:

1. LEX Overview:
- LEX is a program generator designed for lexical processing of text. It reads a set of regular expressions as input and generates a C program that recognizes lexical patterns defined by those regular expressions.

2. Lexical Analyzer Generation:
- Users specify patterns using regular expressions along with corresponding actions in a LEX source file. LEX then generates a C program that functions as a lexical analyzer. The generated program recognizes patterns in the input text and performs specified actions when a pattern is matched.

3. Syntax of a LEX Source File:
- A LEX source file consists of a series of regular expressions and associated C code fragments. The structure typically includes rules and optional C code sections.

```lex
%%
pattern1     action1
pattern2     action2
...
%%
additional C code
```

4. Integration with Yacc:
- LEX is often used in conjunction with Yacc. While LEX handles the lexical analysis phase, Yacc deals with parsing. Together, they form a powerful combination for generating compilers.

5. Example LEX Program:
- Here's a simple example of a LEX program that recognizes basic arithmetic operators and numbers:

```lex
%%
"+"     printf("Addition Operator\n");
"-"     printf("Subtraction Operator\n");
"*"     printf("Multiplication Operator\n");
"/"     printf("Division Operator\n");
[0-9]+  printf("Integer Literal: %s\n", yytext);
.       printf("Unrecognized Character\n");
%%

int main() {
    yylex();
    return 0;
}
```

This example defines patterns for arithmetic operators and integers. When executed, it will print messages based on the recognized patterns.

6. Generating and Compiling:

- To generate a lexical analyzer from a LEX source file, you use the LEX tool. For example, if your LEX source file is named `lexer.l`, you might run:

```bash
lex lexer.l
```

This produces a C file (`lex.yy.c`). You can then compile this C file and link it to create an executable.

```bash
cc lex.yy.c -o lexer -ll
```

The `-ll` flag links the program with the LEX library.

7. Executing the Lexical Analyzer:

- After compilation, you can run the generated lexer:

```bash
./lexer
```

This is a basic overview of LEX and its integration with the compiler construction process. Combining LEX with Yacc allows developers to create complete compilers for programming languages.

1.8 Formal Grammars and their Application to Syntax Analysis

Formal grammars play a crucial role in the field of syntax analysis, which is a phase in compiler construction responsible for analysing the syntactic structure of a programming language's source code. These grammars provide a formal and precise way to describe the syntax or structure of a language. The two most commonly used types of formal grammars in syntax analysis are context-free grammars (CFGs) and Backus-Naur Form (BNF) notation.

1. Context-Free Grammars (CFGs):

- Context-Free Grammars are a type of formal grammar introduced by Noam Chomsky. A CFG consists of a set of production rules that define how strings in the language can be generated. A production rule typically consists of a non-terminal symbol (representing a syntactic category) and a sequence of terminals and/or non-terminals.

Example CFG production rule:
```
Statement -> if (Expression) Statement else Statement
```

2. BNF Notation:
- Backus-Naur Form (BNF) is a metalanguage used to formally describe the syntax of programming languages. BNF notation uses a set of production rules, where each rule defines the syntax of a specific language construct. BNF is commonly used to describe the syntax of programming languages in language specifications.

Example BNF rule:
```
<Statement> ::= if (<Expression>) <Statement> else <Statement>
```

Applications to Syntax Analysis:

1. Language Specification:
- Formal grammars are used to specify the syntax of a programming language in a clear and unambiguous way. Language designers define the rules for valid programs using production rules, helping ensure consistency and clarity in language specifications.

2. Parsing:
- The primary application of formal grammars in syntax analysis is in parsing, the process of analyzing the syntactic structure of source code. Parsing involves determining whether a given program adheres to the rules specified by the grammar. There are various parsing algorithms, such as LL parsers, LR parsers, and recursive descent parsers, designed to work with different types of grammars.

3. Compiler Construction:
- Formal grammars are a fundamental component in compiler construction. Compiler front ends use grammars to define the syntactic structure of programming languages. Lexical analyzers and parsers are generated based on these grammars to recognize tokens and parse the source code.

4. Error Detection and Reporting:
- Formal grammars facilitate the detection of syntax errors in source code. When the parser encounters a sequence that does not conform to the grammar rules, it can report a syntax error. The use of formal grammars helps in providing meaningful error messages to the programmer.

5. Language Transformation:
- Formal grammars are used in language transformation tools to manipulate and transform source code. Tools like compiler optimizers may use formal grammars to recognize and apply specific transformations to improve code efficiency.

6. IDE Features:
- Integrated Development Environments (IDEs) leverage formal grammars to provide features like syntax highlighting, code completion, and error checking. These features

enhance the developer's experience by providing real-time feedback on the syntactic correctness of their code.

In summary, formal grammars, particularly context-free grammars and BNF notation, are foundational tools in syntax analysis and compiler construction. They enable the formal specification of programming language syntax, aid in parsing source code, and play a crucial role in various aspects of language processing and tool development.

1.9 BNF Notation

Backus-Naur Form (BNF) is a widely used notation in the field of compiler construction and formal language specification. It is a metalanguage used to formally describe the syntax of programming languages, as well as various other formal languages. BNF notation is particularly common in the context of context-free grammars and language specifications. Here's an overview of how BNF notation is used in compiler construction:

1. Syntax Rule Definitions:
- BNF is employed to define syntax rules for a language. These rules describe the valid structure and composition of programs written in the language. Each rule typically defines how a language construct can be composed using other constructs.

Example BNF rule for a simple assignment statement:
```
<AssignmentStatement> ::= <Identifier> "=" <Expression> ";"
```

This rule states that an `<AssignmentStatement>` consists of an `<Identifier>`, followed by "=", followed by an `<Expression>`, and terminated by ";".

2. Terminal and Non-terminal Symbols:
- In BNF, symbols are categorized into terminal and non-terminal symbols. Terminal symbols represent actual language constructs, while non-terminal symbols are placeholders representing groups of language constructs.

Example:
```
<Identifier> ::= [a-zA-Z][a-zA-Z0-9]*
```

Here, `<Identifier>` is a non-terminal symbol, and the actual identifiers are represented by the regular expression `[a-zA-Z][a-zA-Z0-9]*`.

3. Alternatives and Optional Elements:
- BNF allows the specification of alternatives and optional elements using symbols like "|" (pipe) and brackets.

Example:
```
```

```
<Statement> ::= <AssignmentStatement> | <IfStatement> | <WhileStatement>
```

This rule states that a `<Statement>` can be an `<AssignmentStatement>`, an `<IfStatement>`, or a `<WhileStatement>`.

4. Repetition:
- BNF provides a way to express repetition using symbols like "*" (asterisk) and "+" (plus).

Example:
```
<FunctionCall> ::= <Identifier> "(" <ExpressionList>? ")"
<ExpressionList> ::= <Expression> ("," <Expression>)*
```

The `<ExpressionList>?` indicates that the expression list is optional, and `("," <Expression>)*` indicates that there can be zero or more expressions separated by commas.

5. Recursive Definitions:
- BNF allows for recursive definitions, which are useful for expressing nested structures in a language.

Example:
```
<Expression> ::= <Term> "+" <Expression>
               | <Term> "-" <Expression>
               | <Term>
```

This recursive definition represents arithmetic expressions with addition and subtraction operations.

6. Language Specification:
- BNF is often used to create a formal specification of a programming language. This specification serves as the foundation for the design and implementation of compilers for the language.

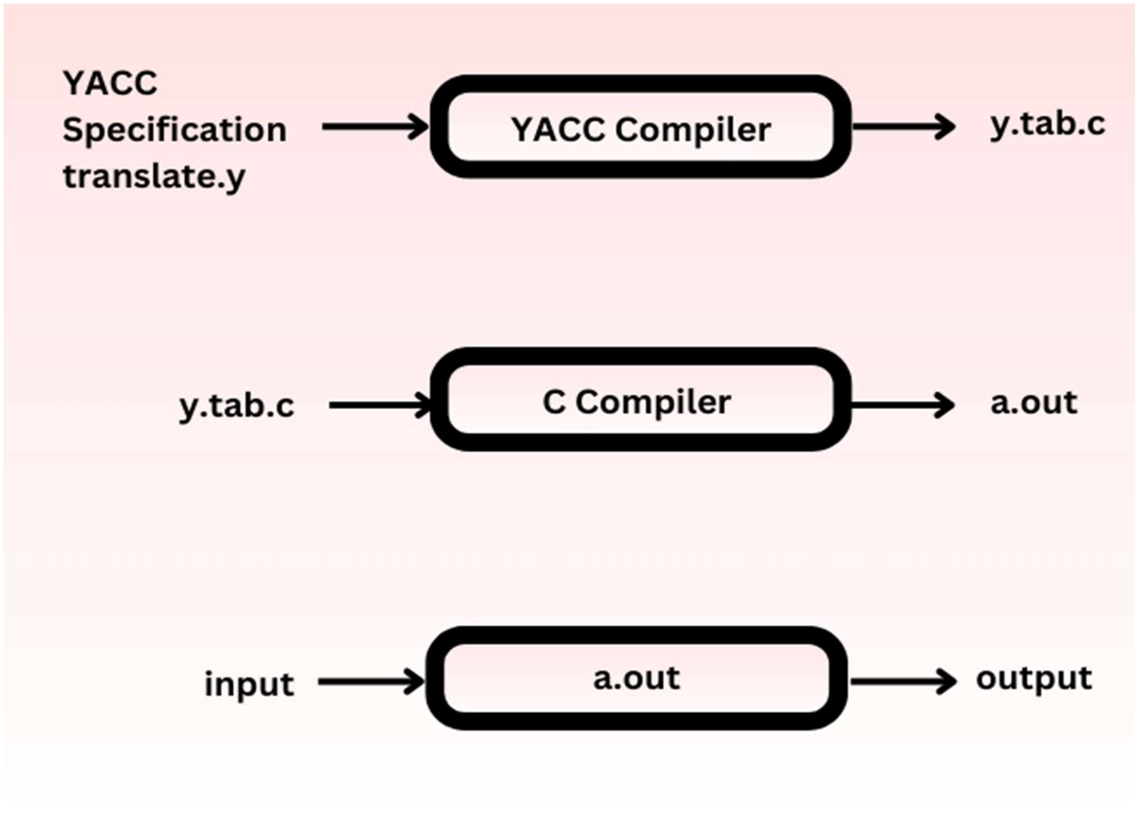
Example excerpt from a language specification:
```
<Program> ::= <Statement>*
<Statement> ::= <AssignmentStatement> | <IfStatement> | <WhileStatement>
<AssignmentStatement> ::= <Identifier> "=" <Expression> ";"
...
```

7. Automated Tools:
- BNF specifications can be used by automated tools, such as parser generators like Yacc or Bison, to generate parsers for a language. These tools take BNF rules as input and produce code for parsing and analyzing the syntax of programs written in the specified language.

In summary, BNF notation is a powerful and widely used tool in compiler construction for formally specifying the syntax of programming languages. It provides a clear and concise way to define the structure of language constructs, aiding in the development of parsers and compilers.

1.10 Yacc (Yet Another Compiler Compiler)

YACC(Yet Another Compiler Compiler)," serves as a powerful grammar parser and generator. In essence, it functions as a tool that takes in a grammar specification and transforms it into executable code capable of meticulously structuring input tokens into a coherent syntactic tree, aligning seamlessly with the prescribed grammar rules.
Stephen C. Johnson developed YACC in compiler design in the early 1970s. Initially, the YACC was written in the B programming language and was soon rewritten in C. It was originally designed for being complemented by Lex.



The input of YACC in compiler design is the rule or grammar, and the output is a C program.

Parts of a YACC Program in Compiler Design

The parts of YACC program are divided into three sections:

```
/* definitions */
 ....
```

```
%%
/* rules */
....
%%


/* auxiliary routines */
....
```

Definitions: these include the header files and any token information used in the syntax. These are located at the top of the input file. Here, the tokens are defined using a modulus sign. In the YACC, numbers are automatically assigned for tokens. Let us see some examples:

```
%token ID
{% #include <stdio.h> %}
```

Rules: The rules are defined between *%%* and *%%.* These rules define the actions for when the token is scanned and are executed when a token matches the grammar.
Auxiliary Routines: Auxiliary routines contain the function required in the rules section. This Auxiliary section includes the *main()* function, where the *yyparse()* function is always called.
This *yyparse()* function plays the role of reading the token, performing actions and then returning to the main() after the execution or in the case of an error.
0 is returned after successful parsing and *1* is returned after an unsuccessful parsing.

The YACC is responsible for converting these sections into subroutines which will examine the inputs. This process is made to work by a call to a low-level scanner and is named Parsing.

Yacc (Yet Another Compiler Compiler) is a tool used in the construction of compilers. It is a code generator that takes as input a formal grammar description, typically specified in Backus-Naur Form (BNF) or a similar notation, and generates a parser in the C programming language. The generated parser is then used to parse and analyze the syntactic structure of programs written in the specified language.

Here are the key features and components associated with Yacc:

1. Parser Generator:
- Yacc is a parser generator that automates the generation of parsers based on formal grammar specifications. It takes a high-level language description and produces a parser in C that can recognize the syntactic structure of programs written in that language.

2. Grammar Specification:
- The input to Yacc is a formal grammar specification, often expressed in BNF notation or a similar context-free grammar notation. This specification defines the syntactic rules and structure of the programming language.

3. Actions and Semantics:
- In addition to defining grammar rules, Yacc allows developers to associate actions with specific grammar rules. These actions are snippets of code written in C and are executed when the corresponding rule is recognized during parsing. Actions are used to perform semantic actions, such as building an abstract syntax tree or generating intermediate code.

4. Tokenization:
- Yacc works in conjunction with lexical analyzer generators like Lex or Flex to tokenize the input source code. Lex generates a lexical analyzer, which breaks the input into tokens, and Yacc's parser then works with these tokens based on the grammar rules.

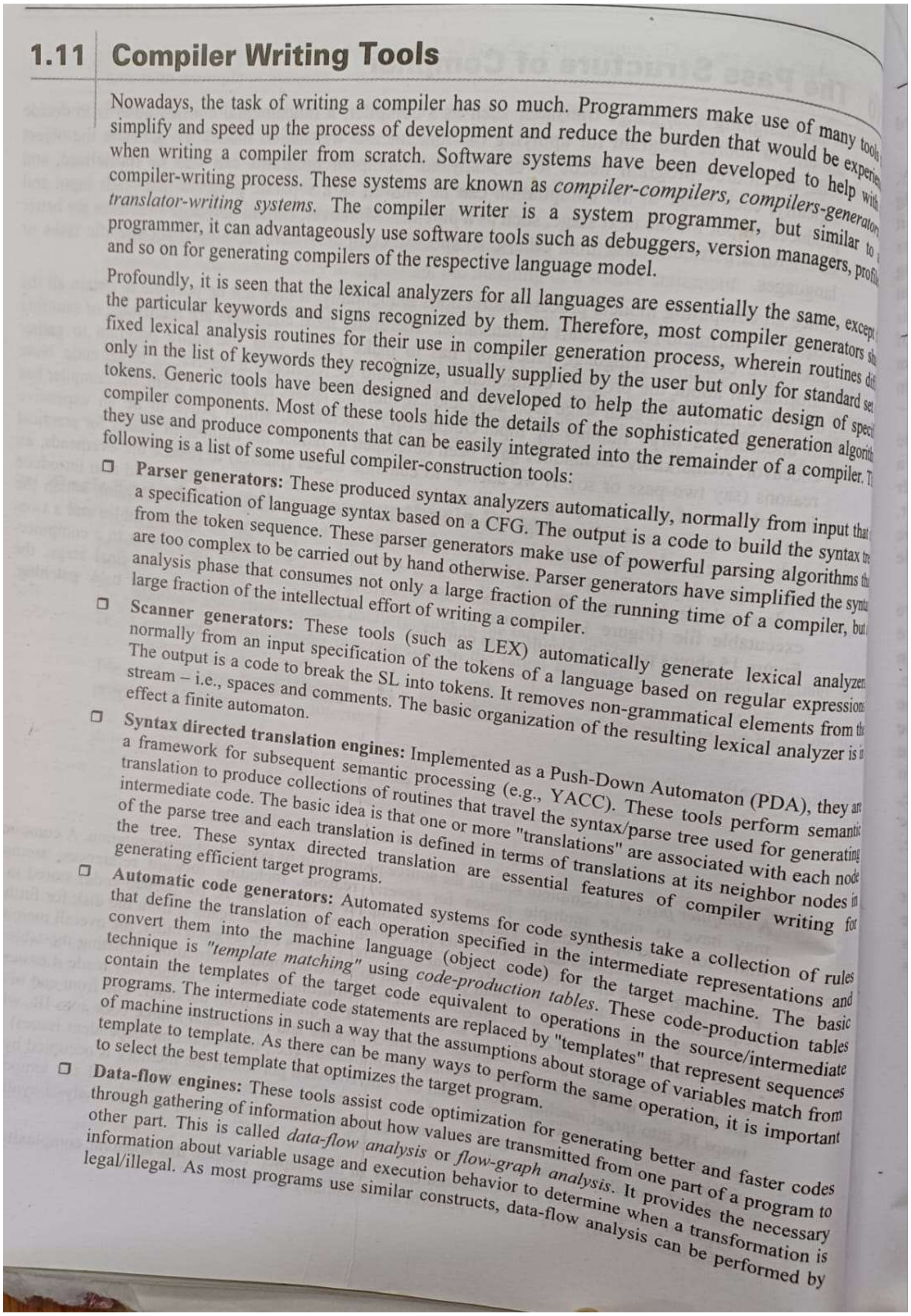
5. Bottom-Up Parsing:
- Yacc generates parsers that use a bottom-up parsing technique known as LALR (Look-Ahead Left-to-Right, Rightmost Derivation) parsing. This parsing strategy efficiently handles a broad class of grammars and allows for the construction of parsers with relatively small tables.

6. Integration with Lex:
- Yacc is often used in combination with Lex (or Flex) to create a complete compiler front end. Lex is used for lexical analysis (tokenization), and Yacc is used for syntactic analysis (parsing). The combination of Lex and Yacc allows for the construction of compilers for various programming languages.

7. Error Handling:
- Yacc-generated parsers typically include error recovery mechanisms. When a syntax error is encountered, the parser attempts to recover and continue parsing, providing meaningful error messages to aid developers in identifying and fixing syntax errors.

8. Abstract Syntax Trees (AST):
- Yacc can be used to construct abstract syntax trees (ASTs) during parsing. ASTs are hierarchical structures that represent the syntactic structure of a program. These trees are useful for subsequent phases of the compiler, such as semantic analysis and code generation.

In summary, Yacc is a powerful tool in compiler construction that automates the generation of parsers based on formal grammars. It is commonly used in conjunction with lexical analyzer generators to create the front end of compilers for various programming languages.



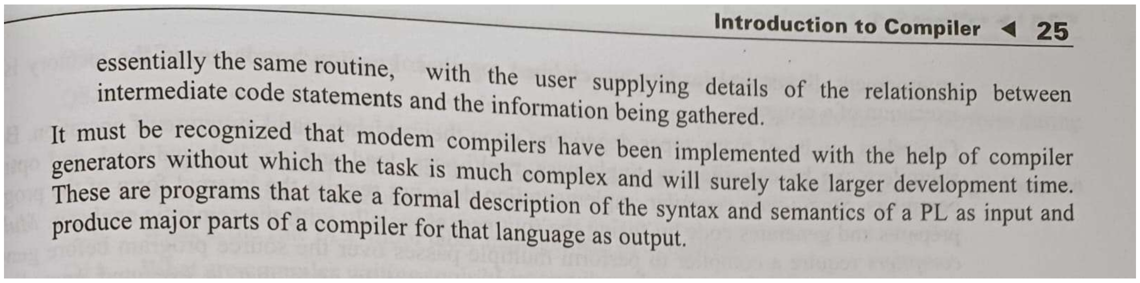
1.11 Compiler Writing Tools

Nowadays, the task of writing a compiler has so much. Programmers make use of many tools simplify and speed up the process of development and reduce the burden that would be experie when writing a compiler from scratch. Software systems have been developed to help with compiler-writing process. These systems are known as *compiler-compilers, compilers-generators translator-writing systems.* The compiler writer is a system programmer, but similar to programmer, it can advantageously use software tools such as debuggers, version managers, profil and so on for generating compilers of the respective language model.

Profoundly, it is seen that the lexical analyzers for all languages are essentially the same, except the particular keywords and signs recognized by them. Therefore, most compiler generators sh fixed lexical analysis routines for their use in compiler generation process, wherein routines dif only in the list of keywords they recognize, usually supplied by the user but only for standard set tokens. Generic tools have been designed and developed to help the automatic design of speci compiler components. Most of these tools hide the details of the sophisticated generation algorit they use and produce components that can be easily integrated into the remainder of a compiler. T following is a list of some useful compiler-construction tools:

- **Parser generators:** These produced syntax analyzers automatically, normally from input that a specification of language syntax based on a CFG. The output is a code to build the syntax tr from the token sequence. These parser generators make use of powerful parsing algorithms th are too complex to be carried out by hand otherwise. Parser generators have simplified the synta analysis phase that consumes not only a large fraction of the running time of a compiler, but large fraction of the intellectual effort of writing a compiler.
- **Scanner generators:** These tools (such as LEX) automatically generate lexical analyzers normally from an input specification of the tokens of a language based on regular expression The output is a code to break the SL into tokens. It removes non-grammatical elements from th stream – i.e., spaces and comments. The basic organization of the resulting lexical analyzer is i effect a finite automaton.
- **Syntax directed translation engines:** Implemented as a Push-Down Automaton (PDA), they are a framework for subsequent semantic processing (e.g., YACC). These tools perform semantic translation to produce collections of routines that travel the syntax/parse tree used for generating intermediate code. The basic idea is that one or more "translations" are associated with each node of the parse tree and each translation is defined in terms of translations at its neighbor nodes in the tree. These syntax directed translation are essential features of compiler writing for generating efficient target programs.
- **Automatic code generators:** Automated systems for code synthesis take a collection of rules that define the translation of each operation specified in the intermediate representations and convert them into the machine language (object code) for the target machine. The basic technique is *"template matching"* using *code-production tables*. These code-production tables contain the templates of the target code equivalent to operations in the source/intermediate programs. The intermediate code statements are replaced by "templates" that represent sequences of machine instructions in such a way that the assumptions about storage of variables match from template to template. As there can be many ways to perform the same operation, it is important to select the best template that optimizes the target program.
- **Data-flow engines:** These tools assist code optimization for generating better and faster codes through gathering of information about how values are transmitted from one part of a program to other part. This is called *data-flow analysis* or *flow-graph analysis*. It provides the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal. As most programs use similar constructs, data-flow analysis can be performed by

]



essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

It must be recognized that modem compilers have been implemented with the help of compiler generators without which the task is much complex and will surely take larger development time. These are programs that take a formal description of the syntax and semantics of a PL as input and produce major parts of a compiler for that language as output.