Compiler Design

Department of Computer Science

Semester – V

Bachelor of Technology(B.Tech)

Faculty Name

Mr. Rohit Pratap Singh
Department of Computer Science & Information Technology
IIIT, Sonipat

UNIT 1

Programming Language

A programming language is an organized way of communicating with a computer using a set of commands and instructions, instructing the computer to perform specific task. It is compiler-based languages. Programming languages run faster compare then scripting languages

1. It creates a .exe file.
2. Need compiler.
3. Complex
4. More code needs

Examples

1. C
2. C++
3. Java
4. Pascal
5. COBOL
6. Basic

COBOL	Common Business-Oriented Language
BASIC	Beginners' All-purpose Symbolic Instruction Code

Scripting Language

A scripting language is a programming language that supports scripts Scripting languages don't require to be compiled rather they are interpreted. Scripting languages are primarily used for web applications. It an Interpreter based language Takes less time to code as it needs less coding.

1. Does not creates a .exe file.
2. NO Need To compiler.
3. Easy to write
4. Less code needs

Examples

1. PHP
2. Python
3. JavaScript
4. VB Script
5. Perl
6. Ruby

PHP	Hypertext Preprocessor
Perl	Practical Extraction and Report Language

Markup language

Markup language is a computer language that uses tags to define elements within a document. No Need To compiler. The language specifies code for formatting, both the layout and style, within a text file.

Examples

1. HTML
2. XML
3. XHTML
4. SGML

HTML	Hypertext Markup Language
XML	Extensible Markup Language

Scripting Language

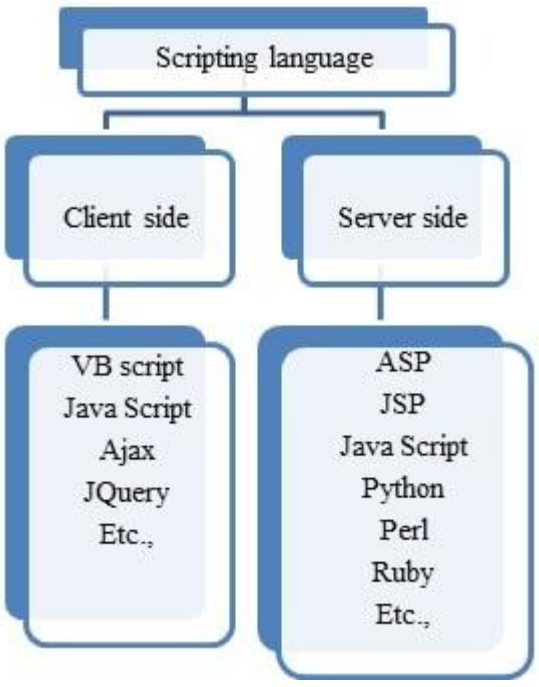
Scripting Language is built into a specific application. Complete application is downloaded to the client browser. Can create dynamic web page.

Scripting language is a programming language that support writing of scripts.

Uses of Scripting Language

1. Interpreter based
2. No need to compile
3. Easy to write
4. Less code required
5. Not Create .exe file
6. Used for Web development

Types of Scripting Language



1. **Client-Side Scripting**
2. **Server-Side Scripting**

Client-side scripting

Client-side scripting cannot used to connect to database on server. It Executed in the web browser.

Uses of Client-side scripting

1. Source code is visible to user
2. Run on computer browser
3. It is faster as compared to server-side scripting
4. Frontend
5. No need interaction with server
6. Insecure
7. Collect User Input

Examples

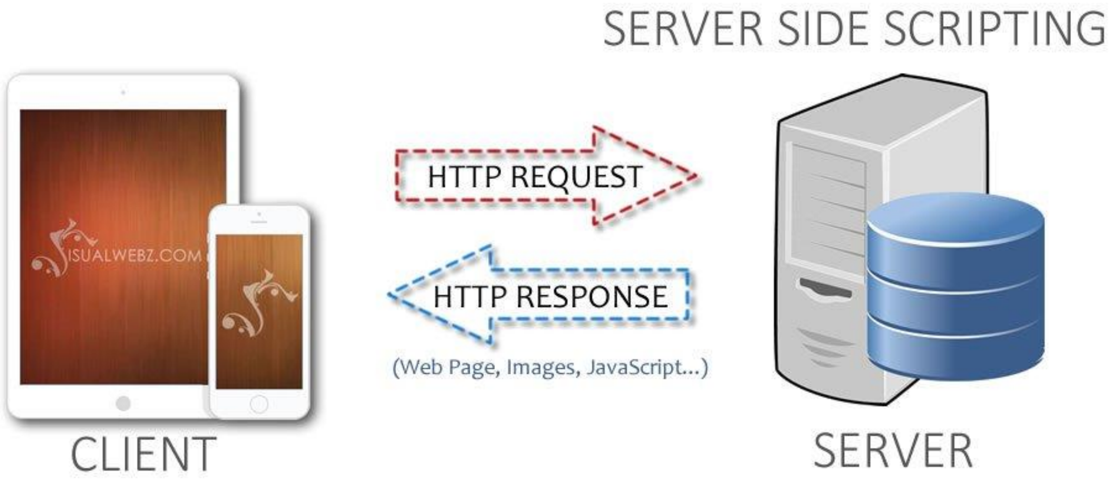
1. JavaScript
2. ActionScript
3. VBScript
4. Ajax
5. HTML
6. CSS

Server-side scripting

Server-side scripting used to connect to database on server. It Executed in the web Server. Generate Content for the dynamic web pages

Uses of Server-side scripting

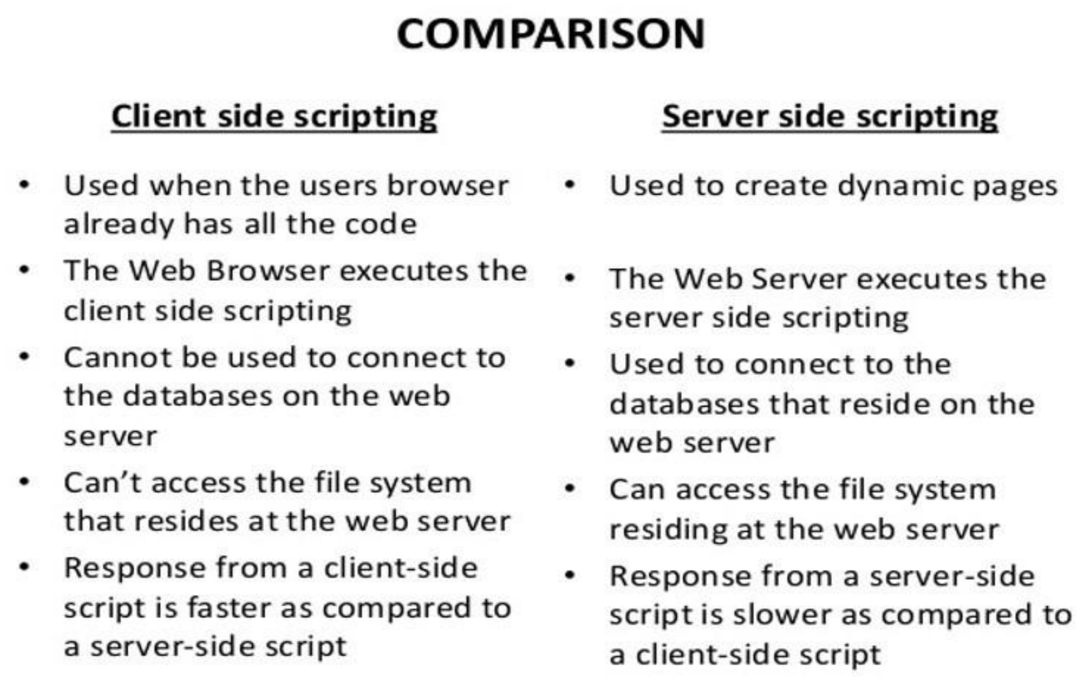
1. Source code is not visible to user
2. Run on web server
3. It is slower scripting language
4. Backend
5. Interaction with server
6. More secure
7. Collect User Input and process data with web server



Examples

1. PYTHON(.py)
2. RUBY(.rb)
3. PHP(.php)
4. ASP(.asp)
5. ASP.NET(.aspx)
6. JSP(.jsp)
7. PERL(.pl)

Client-side scripting Vs Server-side scripting



COMPARISON

<u>Client side scripting</u>	<u>Server side scripting</u>
• Used when the users browser already has all the code	• Used to create dynamic pages
• The Web Browser executes the client side scripting	• The Web Server executes the server side scripting
• Cannot be used to connect to the databases on the web server	• Used to connect to the databases that reside on the web server
• Can't access the file system that resides at the web server	• Can access the file system residing at the web server
• Response from a client-side script is faster as compared to a server-side script	• Response from a server-side script is slower as compared to a client-side script

UNIT 2

Regular Grammar

Regular grammar is a type of grammar that describes a regular language. A regular grammar four Tuple, G = (V, T, P, S)

V: Finite set of non-terminal symbols

T: Finite set of terminal symbols

P: Production rules

S: start symbol.

This grammar can be of two forms:

1. Left Linear Regular Grammar
2. Right Linear Regular Grammar

Left and Right linear Regular Grammars

Left Linear Regular Grammar

All the non-terminals on the right-hand side exist at the left ends.

Example 1

S ⇢ a, S ⇢ Aab, S ⇢ ∈

Where

S and A are non-terminals
a and b are terminal
∈ is empty string

Example 2

S ⇢ B10 | B00

B ⇢ B11 | B1 | 1 | 0

where
S and B are non-terminals
0 and 1 are terminals

Right Linear Regular Grammar

All the non-terminals on the right-hand side exist at the right ends.

Example 1

S ⇢ a, S ⇢ abA, S ⇢∈

where

S and A are non-terminals a and b are terminal

∈ is empty string

Example 2

S ⇢ 10B | 00S

B ⇢ 11B | 1B | 1 | 0

where

S and B are non-terminals 0 and 1 are terminals

Limitations of Finite Automata

1. Finite input.
2. Input tape is read only
3. Finite amount of memory
4. Head movement is in only one direction.

UNIT 1

INTRODUCTION TO COMPILERS AND ITS PHASES

A compiler is a program takes a program written in a source language and translates it into an equivalent program in a target language. The sourcelanguage is a high level language and target language is machine language.

Source program -> COMPILER -> Target program

Necessity of compiler

- Techniques used in a lexical analyzer can be used in text editors, information retrievalsystem, and pattern recognition programs.

- Techniques used in a parser can be used in a query processing system such as SQL.

- Many software having a complex front-end may need techniques used in compiler design.

- A symbolic equation solver which takes an equation as input. That program should parsethe given input equation.

- Most of the techniques used in compiler design can be used in Natural LanguageProcessing (NLP) systems.

Properties of Compiler

1. Correctness
2. Correct output in execution.
3. It should report errors
4. Correctly report if the programmer is not following language syntax.
5. Efficiency
6. Compile time and execution.
7. Debugging / Usability.

Compiler	Interpreter
1. It translates the whole program at atime. 2. Compiler is faster. 3. Debugging is not easy. 4. Compilers are not portable.	1. It translate statement by statement. 2. Interpreter is slower. 3. Debugging is easy. 4. Interpreter are portable.

Types of compiler

Native code compiler

A compiler may produce binary output to run /execute on the same computer and operatingsystem. This type of compiler is called as native code compiler.

1) **Cross Compiler**

A cross compiler is a compiler that runs on one machine and produce object code foranother machine.

2) **Bootstrap compiler**

If a compiler has been implemented in its own language . self-hosting compiler.

3) **One pass compiler**

The compilation is done in one pass over the source program, hence the compilation is completed very quickly. This is used for the programming language PASCAL, COBOL, FORTAN.

4) **Multi-pass compiler (2 or 3 pass compiler)**

In this compiler , the compilation is done step by step . Each step uses the result of theprevious step and it creates another intermediate result.
Example:- gcc , Turboo C++

5) **JIT Compiler**

This compiler is used for JAVA programming language and Microsoft .NET

6) **<u>Source to source compiler</u>**

It is a type of compiler that takes a high level language as a input and its output as highlevel language. Example Open MP
List of compiler

1. Ada compiler
2. ALGOL compiler
3. BASIC compiler
4. C# compiler
5. C compiler
6. C++ compiler
7. COBOL compiler
8. Smalltalk comiler
9. Java compiler

ASSEMBLER

1. It translates assembly language code into machine understandable language.
2. Assembly language is in between the high level languages and machine language.
3. It is also called low level language.
4. This language is not easily readable and understandable by the programmer

Source-to-source Compiler

Source code of one programming language is translated into the source of another language.

Loader

A loader is a program that places programs into memory and prepares them for execution. loader is a part of the OS, which performs the tasks of loading executable files into memory and run them

Compiler Construction Tools

These tools use specific language or algorithm for specifying and implementing the component of the compiler.

1. **Parser generators.**
   Input: Grammatical description of a programming language
   Output: Syntax analyzers.
   Produces a syntax analyzer.

2. **Scanner generators.**
   Input: Regular expression description of the tokens of a language
   Output: Lexical analyzers.
   Produces a Lexical analyzer.
3. **Syntax-directed translation engines.**

   Input: Parse tree.
   Output: Intermediate code.

   Generates intermediate code.

4. **Automatic code generators**
   Input: Intermediate language.
   Output: Machine language.
5. **Data-flow engines**

Various phases of a compiler

There are two major parts of a compiler: Analysis and Synthesis

In analysis phase

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer.

In synthesis phase

1. Intermediate Code Generator
2. Code Generator
3. Code Optimizer

1.Lexical Analysis

Lexical analyzer phase is the first phase of compilation process. It takes source code as input.

2.Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output.

3.Semantic Analysis

Semantic analysis is the third phase of compilation process.

4.Intermediate Code Generation

Compiler generates the source code into the intermediate code.

5.Code Optimization

6.Code Generation

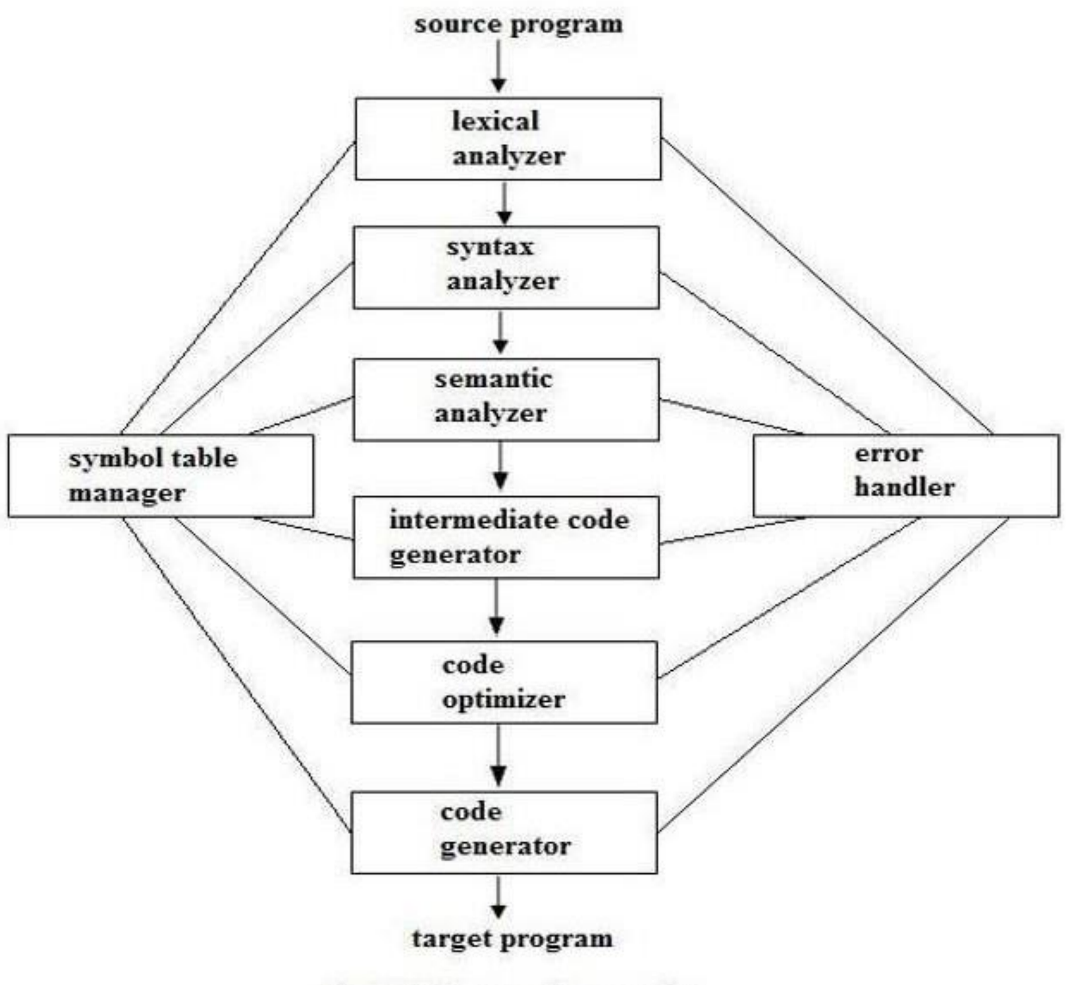


Fig 1.5 Phases of a compiler

Lexical Analyzer

Lexical Analyzer reads the source program character by character and returns the tokens of the source program.

Syntax Analyzer

1. A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program.
2. A syntax analyzer is also called a parser.
3. A parse tree describes a syntactic structure
4. The syntax of a language is specified by a context free grammar (CFG).

Semantic Analyzer

1. A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
2. Type-checking is an important part of semantic analyzer.
3. Normally semantic information cannot be represented by a context-free language used in syntax analyzers

Symbol table

Symbol table information is used by the analysis and synthesis phases.

Essential data structure in compiler.

It is used to verify if a variable has been declared.

It is used to determine the scope of a name.

Regular definition

Defining a pattern for finite strings of symbols. language defined by regular grammar is known as regular language

Properties of Regular Languages

Union

If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular

Complement

If L(G) is regular language, its complement L’(G) will also be regular.

$L(G) = \{a^n \mid n > 1\}$
$L'(G) = \{a^n \mid n <= 1\}$

Kleene Closure

If L1 is a regular language, its Kleene closure L1* will also be regular.

$L1 = (a \cup b)$
$L1^* = (a \cup b)^*$

Concatenation

If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular

Intersection

If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular.

Precedence

1.* highest precedence.

2.Concatenation (.) second-highest precedence.

3. | (Union operator) lowest precedence.

Example

Σ = {*a*, b}

a* (e, a,aa, aaa, aaaa …)
a+ (a, aa, aaa, aaaa …).

L* = {Empty, a, b, aa, ab, ba, bb, aab, aba, aaba, … }

L^+ = {a, b, aa, ab, ba, bb, aab, aaba}

L^2 = {aa, ab, bb, ba}

L^3 = {aaa, aab, bbb, bba ,..}

L^4 = {aaaa, aabb, bbbb, bbaa,…….}

Lexical analysis

lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis

It converts the High-level input program into a sequence of Tokens**.**

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Role of Lexical Analyser

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

It helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.

1. Error can be detected.

2.Error is found during the execution of the program.

3. Removes white spaces and comments from the source program.

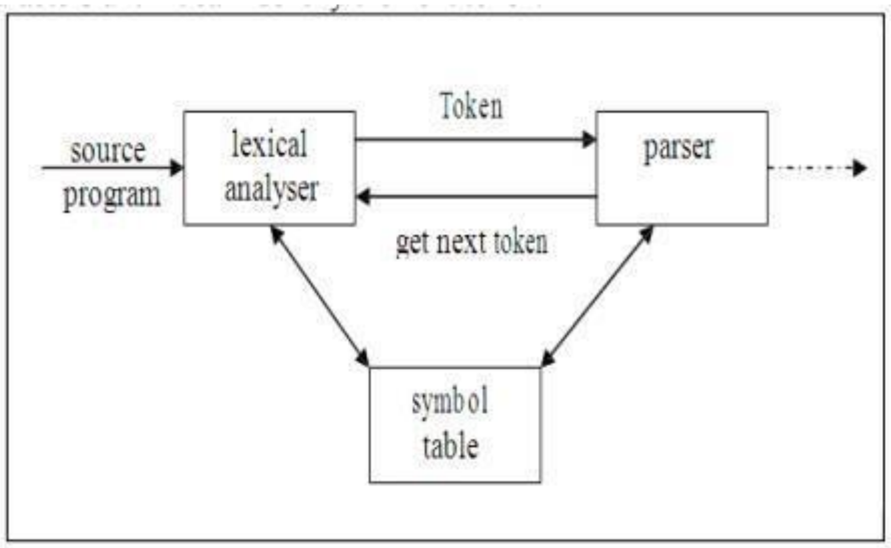


Fig. 1.8 Interaction of lexical analyzer with parser

Difficulties (Issues) in Lexical Analysis

Why separating lexical analysis from parsing

1) Simpler design
2) Compiler efficiency is improved.
3) Compiler portability is enhanced (Linux to window)

Basic Terminologies

Token

Sequence of characters which represents a unit of information in the source program.

1) Identifiers

2) keywords

3) operators

4) special symbols

5)constants

Example

int a = 9;

where

int- keywords

a- identifier

= operator

9 constants

; special symbol

Solution

Token=5

Non-Token

1. Comments
2. Blanks
3. New line

Lexeme

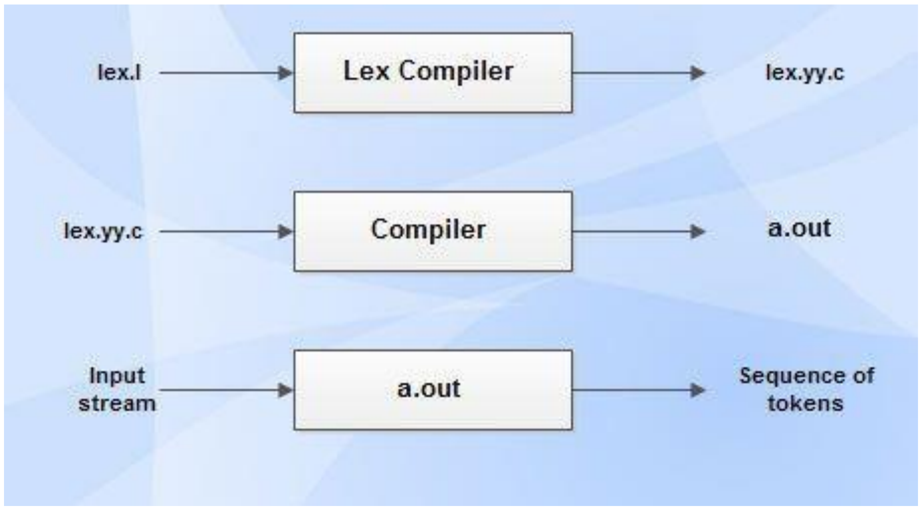
Sequence of characters in the source program that is matched by the pattern for a token.

Pattern

A set of strings in the input for which the same token is produced as output.

LEX (Lexical Analyzer Generator)

1. Program that generates lexical analyzer.
2. It is used with YACC parser generator.
3. Lex tool itself is a lex compiler.
4. Flex and Bison both are more flexible than Lex and Yacc and produces faster code.

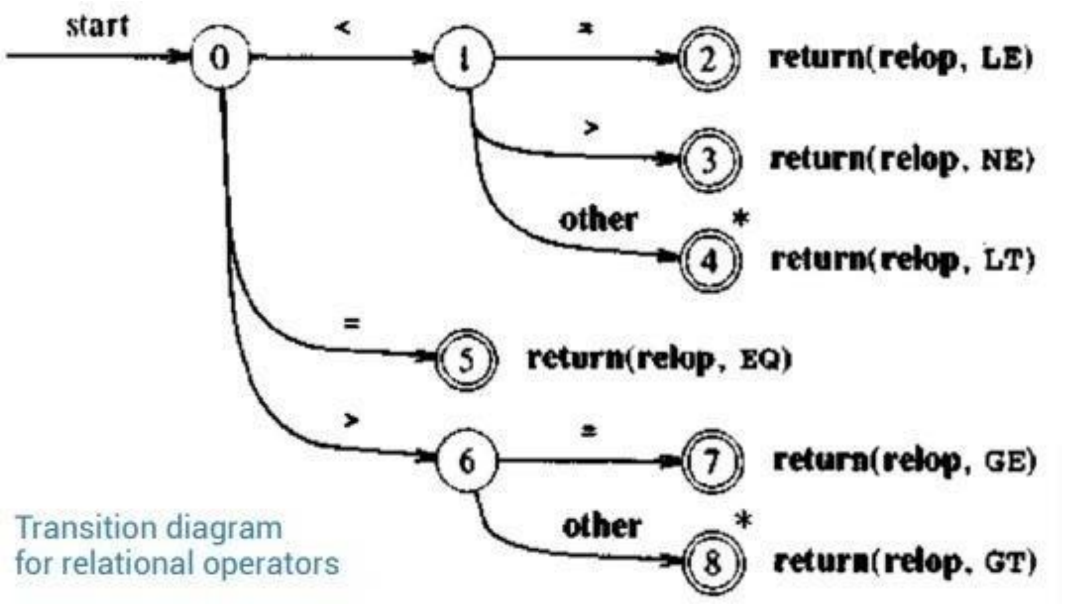


Transition diagrams

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols

Lexeme	Token Name	Attribute Value
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE



Transition diagram
for relational operators

UNIT 2

Terminologies

1. Alphabet

An alphabet is any finite set of symbols

Example − ∑ = {a, b, c, d, e} is an alphabet set

2. String

A string is a finite sequence of symbols taken from ∑.

Example

{0,1} is a valid string on the alphabet set

3. Length of a String

It is the number of symbols present in a string.

Examples −

- If S = ‘caeda’, |S|= 5
- If S = ‘010111’, |S|= 6
- If |S|= 0, it is called an empty string

Language

It can be finite or infinite.

Example

∑ = {a, b}, then L = {ab, aa, ba, bb}

Syntactic specification of Programming Languages

1. **Non-terminals** (also known as **variables**) represent the set of strings in a language.

Examples
A,D,E,F,G

2. **Terminals** represent the symbols of the language.

Examples
A,d,e,f,g (small letters)

3. **Null String**

NIL, ∈

Context free grammar

G= (V, T, P, S)

V Non-terminal symbols

T Terminal symbols.

P Production rules

S Start symbol

Derivation Tree

- **Root vertex −** Start symbol.
- **Vertex −** Non-terminal symbol.
- **Leaves −** Terminal symbol

Derivation Tree Approaches

There are two different approaches to draw a derivation tree

1. Top-down Approach
2. Bottom-up Approach

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Leftmost and Rightmost Derivation

- **Leftmost derivation −** Production applying leftmost variable in each step.
- **Rightmost derivation −** Production applying rightmost variable in each step

Grammar Ambiguity

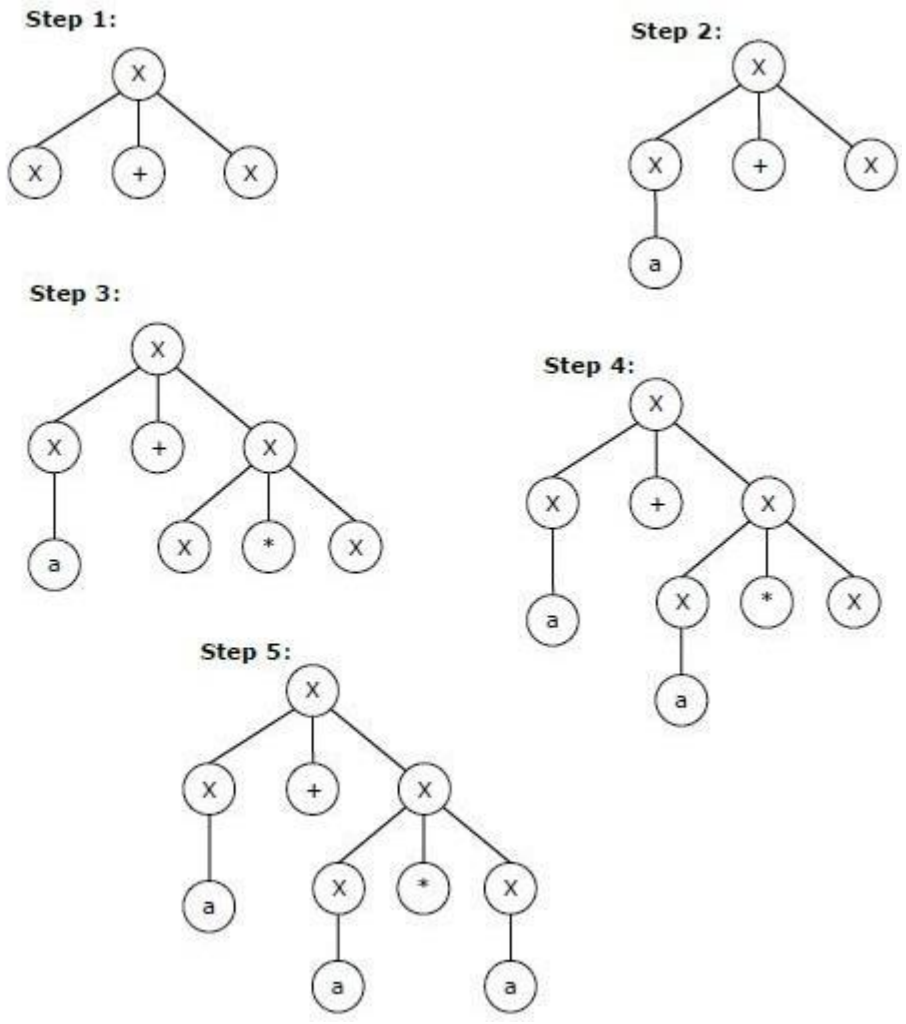
1. More than one leftmost derivation
2. More than one rightmost derivation
3. More than one derivation tree or parse tree

Example 1

X → X+X | X*X |X| a

Find out leftmost derivation string "a+a*a"

X → X+X → a+X → a + X*X → a+a*X → a+a*a

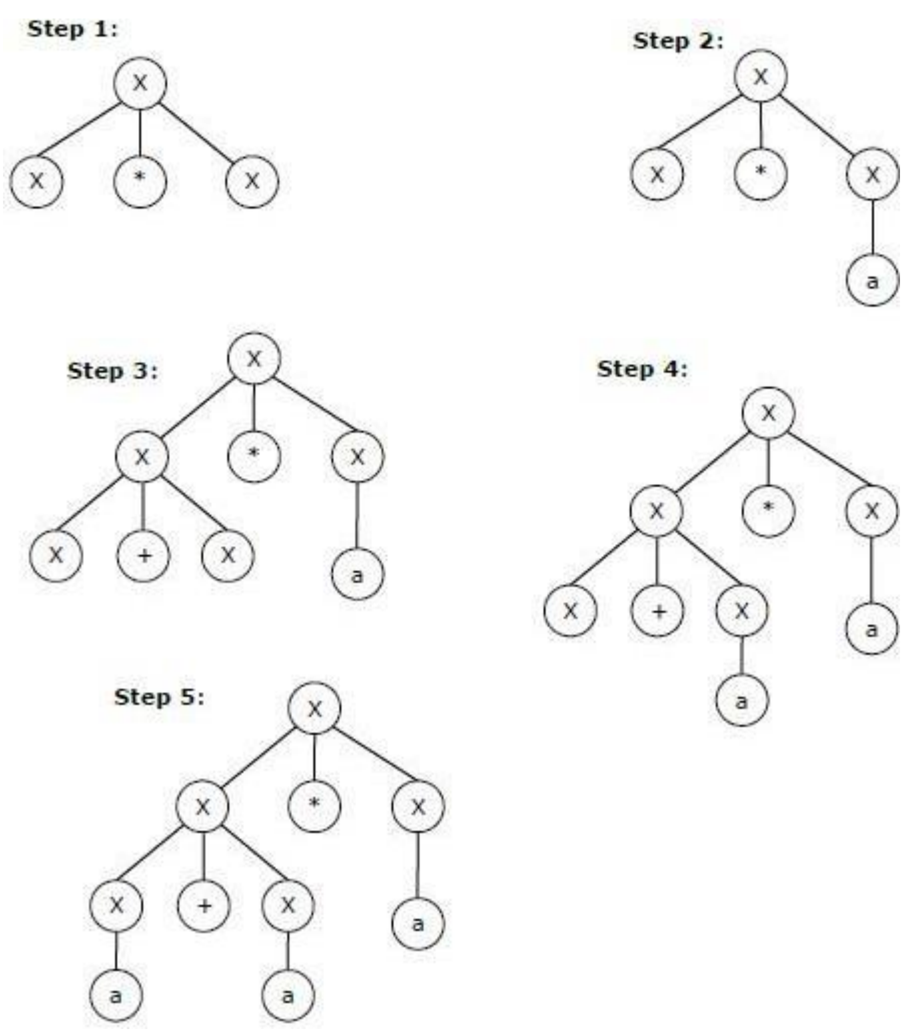


Example 2

X → X+X | X*X |X| a

Find out rightmost derivation string "a+a*a"

X → X*X → X*a → X+X*a → X+a*a → a+a*a



Context Free Language

context-free language (CFL) is a language generated by a context-free grammar (CFG). Context Free Language is accepted by a Pushdown automaton.

Example

$L=\{a^nb^n\}$

Properties of context free grammar

1.Union Operation

The context free languages are closed under union. L1 and L2 are two context free languages.

Example

L1 ∪ L2 is also a context free language.

2.Concatination

Context free languages are closed under concatenation. L1 and L2 are two context free languages.

Example

L1.L2 is also a context free language.

3. Kleene closure

Context free languages are closed under kleen closure. L1 and L2 are two context free languages.

Example

L1* and L2* are also context free languages.

4. Intersection

context free languages are not closed under intersection. L1 and L2 are two context free languages.

Example

L1 ∩ L2 is not a context free language.

5. Complement

Context free languages are not closed under complement. L1 and L2 are two context free languages.

Example

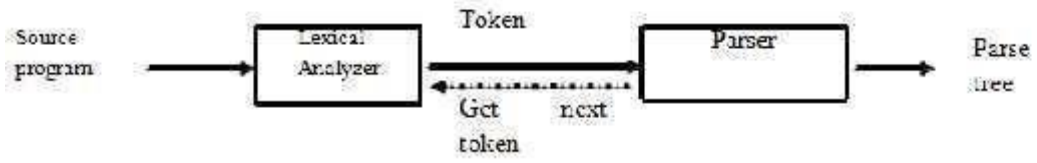
L1′ and L2′ are not context free languages.

Application of context free grammars

1. Parsers constructing syntax tree
2. Describing arithmetic expressions
3. Construction of compilers
4. HTML (Markup Languages)
5. XML (Extensible Markup Languages)

Parser

- Parser works on a stream of tokens.
- The smallest item is a token.



Syntax analysis

Syntax Analyzer creates the syntactic structure of the given source program.

This syntactic structure is mostly a parse tree.

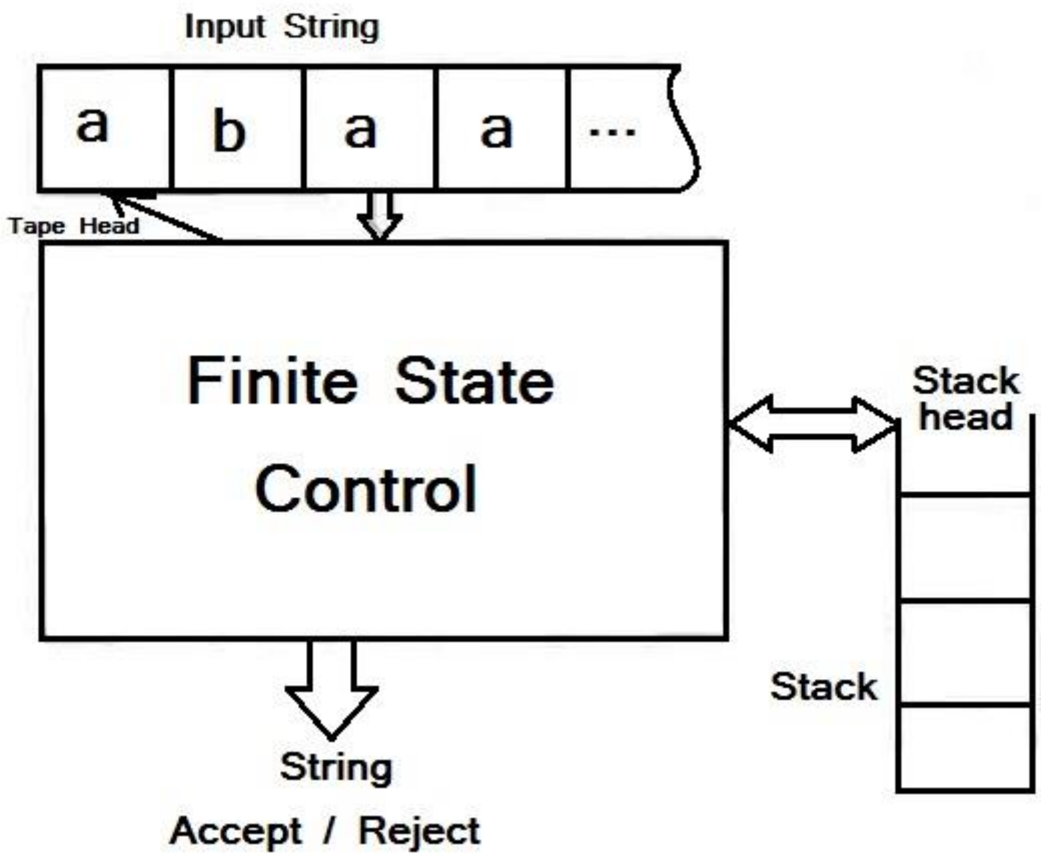
Syntax Analyzer is also known as parser.

The syntax of a programming is described by a context-free grammar (CFG). We will use BNF(Backus-Naur Form) notation in the description of CFGs.

The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.

Stack Operations

- **Push** − New symbol is added at the top.
- **Pop** − Top symbol is read and removed.



UNIT 2

Regular Grammar

Regular grammar is a type of grammar that describes a regular language. A regular grammar four Tuple, G = (V, T, P, S)

V: Finite set of non-terminal symbols

T: Finite set of terminal symbols

P: Production rules

S: start symbol.

This grammar can be of two forms:

1. Left Linear Regular Grammar
2. Right Linear Regular Grammar

Left and Right linear Regular Grammars

Left Linear Regular Grammar

All the non-terminals on the right-hand side exist at the left ends.

Example 1

S ⇢ a, S ⇢ Aab, S ⇢ ∈

Where

S and A are non-terminals
a and b are terminal
∈ is empty string

Example 2

S ⇢ B10 | B00

B ⇢ B11 | B1 | 1 | 0

where
S and B are non-terminals
0 and 1 are terminals

Right Linear Regular Grammar

All the non-terminals on the right-hand side exist at the right ends.

Example 1

S ⇢ a, S ⇢ abA, S ⇢∈

where

S and A are non-terminals a and b are terminal

∈ is empty string

Example 2

S ⇢ 10B | 00S

B ⇢ 11B | 1B | 1 | 0

where

S and B are non-terminals 0 and 1 are terminals

Limitations of Finite Automata

1. Finite input.
2. Input tape is read only
3. Finite amount of memory
4. Head movement is in only one direction.

Properties of finite state machine

1. Flexible
2. Low processor overhead
3. Very simple
4. Creating sequential logic

Application of finite automata

1. Text processing
2. Hardware design
3. Compiler's design

Context free grammar

G= (V, T, P, S)

V Non-terminal symbols

T Terminal symbols.

P Production rules

S Start symbol

Derivation Tree

- **Root vertex** − Start symbol.
- **Vertex** − Non-terminal symbol.
- **Leaves** − Terminal symbol

Derivation Tree Approaches

There are two different approaches to draw a derivation tree

1. Top-down Approach
2. Bottom-up Approach

Top-down Approach −

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach −

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Leftmost and Rightmost Derivation

- **Leftmost derivation −** Production applying leftmost variable in each step.
- **Rightmost derivation −** Production applying rightmost variable in each step

Grammar Ambiguity

1. More than one leftmost derivation
2. More than one rightmost derivation
3. More than one derivation tree or parse tree

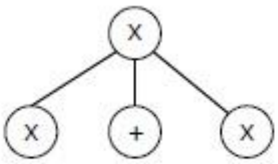
Example 1

X → X+X | X*X |X| a

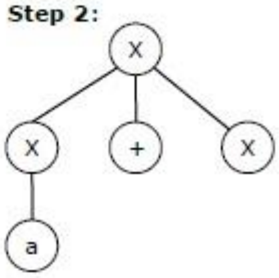
Find out leftmost derivation string "a+a*a"

X → X+X → a+X → a + X*X → a+a*X → a+a*a

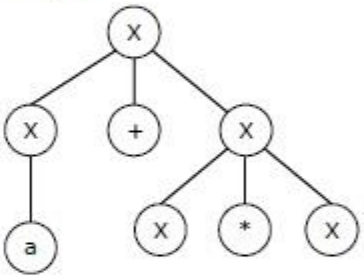
Step 1:



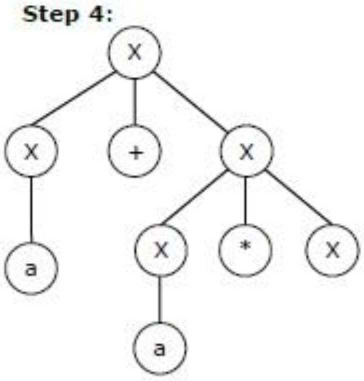
Step 2:



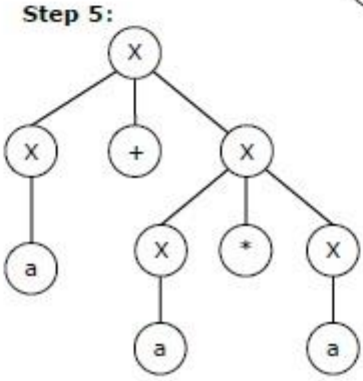
Step 3:



Step 4:



Step 5:



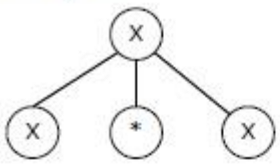
Example 2

X → X+X | X*X |X| a

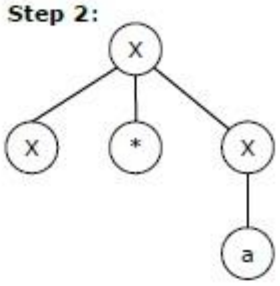
Find out rightmost derivation string "a+a*a"

X → X*X → X*a → X+X*a → X+a*a → a+a*a

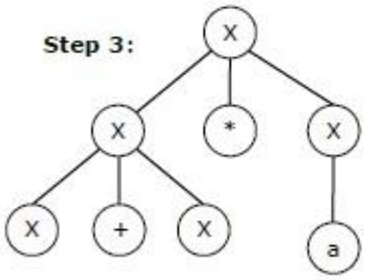
Step 1:



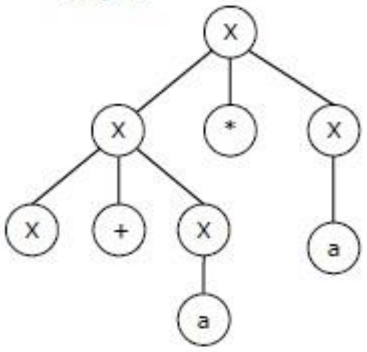
Step 2:



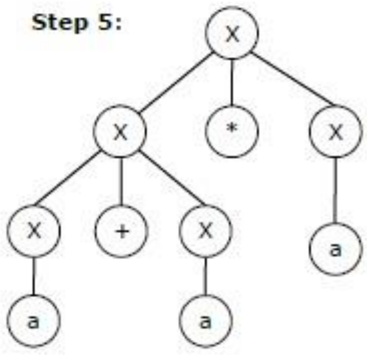
Step 3:



Step 4:



Step 5:



Context Free Language

context-free language (CFL) is a language generated by a context-free grammar (CFG). Context Free Language is accepted by a Pushdown automaton.

Example

L={$a^n b^n$}

Properties of context free grammar

1.Union Operation

The context free languages are closed under union. L1 and L2 are two context free languages.

Example

L1 ∪ L2 is also a context free language.

2.Concatination

Context free languages are closed under concatenation. L1 and L2 are two context free languages.

Example

L1.L2 is also a context free language.

3. Kleene closure

Context free languages are closed under kleen closure. L1 and L2 are two context free languages.

Example

L1* and L2* are also context free languages.

4. Intersection

context free languages are not closed under intersection. L1 and L2 are two context free languages.

Example

L1 ∩ L2 is not a context free language.

5. Complement

Context free languages are not closed under complement. L1 and L2 are two context free languages.

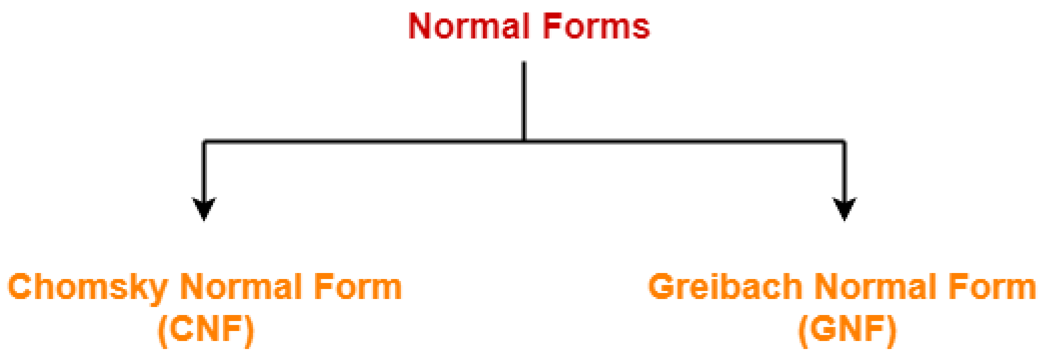
Example

L1′ and L2′ are not context free languages.

Application of context free grammars

1. Parsers constructing syntax tree
2. Describing arithmetic expressions
3. Construction of compilers
4. HTML (Markup Languages)
5. XML (Extensible Markup Languages)

Normal Form



Chomsky Normal Form (CNF)

CNF stands for Chomsky normal form. A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions

- A non-terminal generating two non-terminals
  A →BC
- A non-terminal generating a terminal
  A → a.
- Start symbol generating ε
  A → ε.

where A, B, and C are non-terminals and **a** is terminal.

Example 1

G1= {A → a, A → BC, B → a, C→ a}

Grammar G1 is in CNF as production rules satisfy the rules specified for CNF

Example 2

G2= {A → a, A → aC, C→ a}

Grammar G2 is in CNF as production rules not satisfy the rules specified for CNF

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. Every context-free grammar can be transformed into an equivalent grammar in Greibach normal form

Context-free grammar is in Greibach normal form (GNF) if the right-hand sides of all production rules start with a terminal symbol.

Context free grammar is in Chomsky normal form if all production rules satisfy one of the following conditions:

- A start symbol generating ε

  A → ε.

- A non-terminal generating a terminal which is followed by any number of non-terminals
  A → aBCD
- A non-terminal generating a terminal
  A → a
  B→ b
  C → c
  D → d.

Example 1

G1= {A → a, A → bC, C→ a, A → ε}

The grammar G1 is in GNF as production rules satisfy the rules specified for GNF.

Example 2

G2= {A → a, A → BC, C→ a, A → ε, B→ b}

The grammar G2 is in GNF as production rules not satisfy the rules specified for GNF.

CYK algorithm

Cocke–Younger–Kasami-Algorithm (CYK or CKY) is a highly efficient parsing algorithm

for context-free grammars. CYK used to decide whether a given string belongs to the language of grammar or not.

- CYK operates only on context-free grammars given in Chomsky normal form (CNF).
- Bottom-Up Approach

Decision properties of Regular Languages

1. **Closure Properties**

   Union
   Concatenation
   Intersection
   Complementation
   Homomorphism
   Kleene Closure
   Reversal

2. **Decision Properties**

   Membership
   Emptiness
   Finiteness
   Equivalence

Pumping Lemma for Regular Languages

Pumping lemma in proving that a language is not regular

Pumping lemma is not used for proving whether a language is regular. It is rather used for proving if the language is not regular.

Pumping Lemma

Pumping lemma states that for a regular language L, there exists a constant n such that every string w in L (such that length of w (|w|) >= n) we can break w into three substrings, w = xyz

- For each i >= 0, the string xy^iz is also in L
- |y| >0 {y is not null}
- |xy| <= n

Example

- **Language L = {$a^n b^n$ for n>=0} is not regular.**
- **Language L = {$a^{2n} b^{2n}$for n>=0} is not regular.**
- **L_1 = {$a^n b^n$ | n≥1}, L_2 = {a^n n≥1} then $L_1.L_2$ is not regular.**

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF

Step 2: If the grammar exists left recursion, eliminate it.

Step 3: In the grammar, convert the given production rule into GNF form.

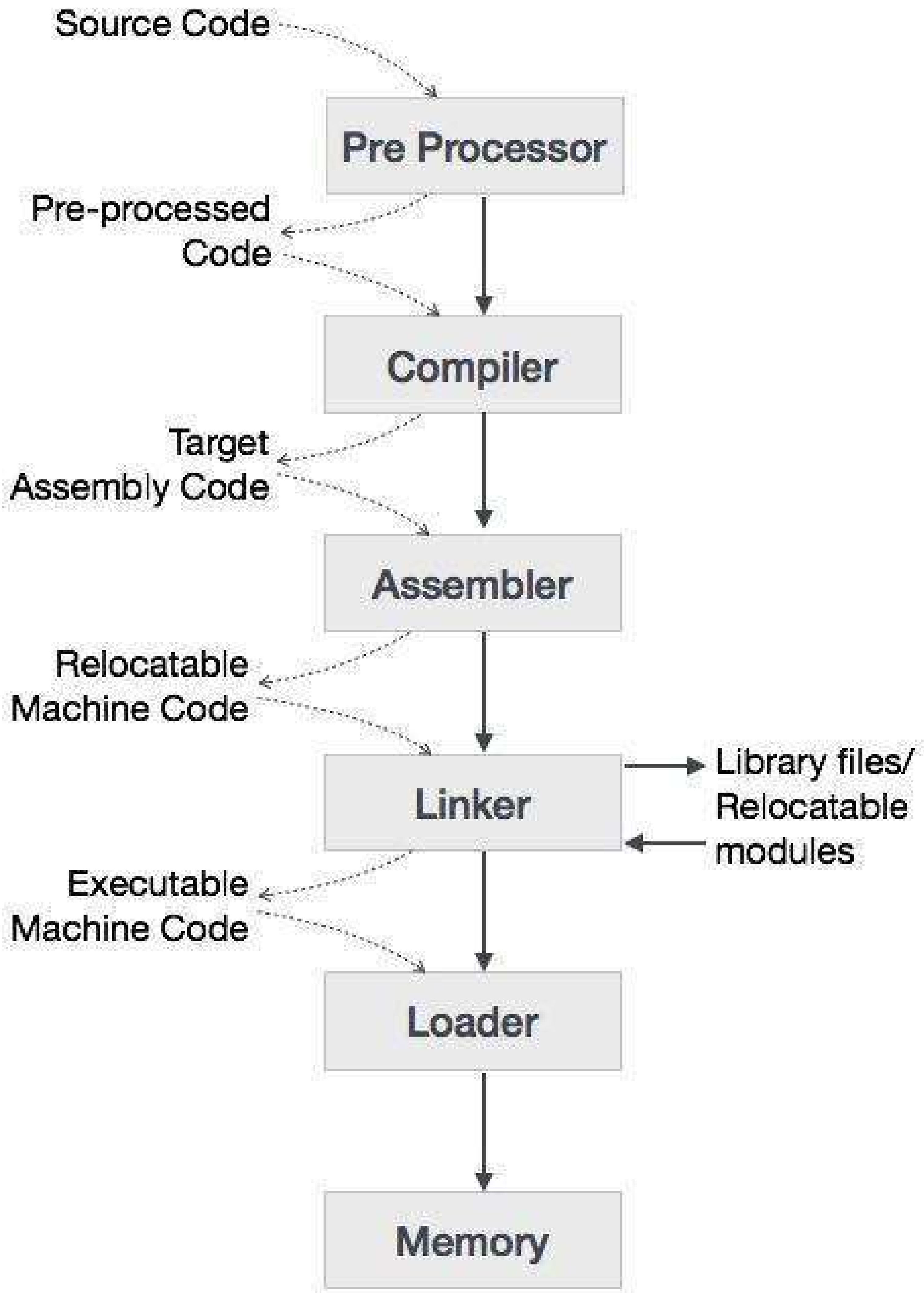
Steps for converting CFG into CNF

Step 1 − If the start symbol **S** occurs on some right side, create a new start symbol **S'** and a new production **S'** → **S**.

Step 2 − Remove Null productions. And Remove unit productions

Step 3. Eliminate terminals from RHS if they exist with other terminals or non-terminals.

Step 4. Eliminate RHS with more than two non-terminals.


Source Code
Pre Processor
Pre-processed Code
Compiler
Target Assembly Code
Assembler
Relocatable Machine Code
Linker
Library files/ Relocatable modules
Executable Machine Code
Loader
Memory