

Hand Book

Compiler Design

Theory

Fall
2020

A BIG ZERO

A Complete Handbook on Compiler Design

BY

Md. Mosfikur Rahman
ID: 181-15-2065

Akash Ahmed
ID: 181-15-1714

Md.kawser Ahamed
ID: 181-15-1722

Md. Abtab Uddin Akib
ID:181-15-2000

Tania sultana khanom
ID: 181-15-1945

Mahfuja Ferdousi Mahin
ID: 181-15-1860

Rakibul Hassan Raja
ID: 181-15-2041

Md. Hasan Imam
ID: 181-15-2027

This handbook presents compiler's working procedure in respectively manual and automatic optimization with C programming.

Supervised By
Mushfikur Rahman
Lecturer
Department of CSE
Daffodil International University



DAFFODIL INTERNATIONAL UNIVERSITY

Dhaka, Bangladesh

DECLARATION

We hereby declare that, this handbook has been made by us under the supervision of Mr. Mushfiqur Rahman, Lecturer, Department of CSE, Daffodil International University. We also declare that neither this handbook nor any part of this handbook has been submitted or published elsewhere.

Supervised by:

Mushfiqur Rahman

Lecturer

Department of CSE

Daffodil International University

Submitted by:

Md. Mosfikur Rahman

ID: 181-15-2065

Department of CSE

Daffodil International University

Akash Ahmed

ID: 181-15-1714

Department of CSE

Daffodil International University

Md. kawser Ahamed

ID: 181-15-1722

Department of CSE

Daffodil International University

Md. Abtab Uddin Akib

ID: 181-15-2000

Department of CSE

Daffodil International University

Tania sultana khanom

ID: 181-15-1945

Department of CSE

Daffodil International University

Mahfuja Ferdousi Mahin

ID: 181-15-1860

Department of CSE

Daffodil International University

Rakibul Hassan Raja

ID: 181-15-2041

Department of CSE

Daffodil International University

Md. Hasan Imam

ID: 181-15-2027

Department of CSE

Daffodil International University

Table of content

Chapter no	Chapter name	Page No.
1	Introduction to Compiler Design Introduction	1
2	Regular Expression	13
3	Context Free Grammars (CFG)	18
4	Non-Deterministic Finite Automata & Deterministic Finite Automata	29
5	Determination of FIRST and FOLLOW Function Parsing	43
6	LR0 Parser and Canonical Table LR0 Parser	46
7	Intermediate Code Generation	51
8	Directed Acyclic Graph (DAG)	58
9	Three Address Code	61
10	Basic Block and Flow Diagram Basic Block	66
11	Code Optimization	73
12	Lab Part	iv

Chapter 1

Introduction to Compiler Design

Introduction:

Computers are a balanced software and hardware combination. Hardware is only a piece of mechanical device and compatible software controls its functions. In the form of electronic charging, which is the equivalent of binary language in software programming, hardware understands instructions. There are only two alphabets in a binary script, 0 and 1. The hardware codes must be written in binary format, which is simply a sequence of 1s and 0s, to instruct the computer. For computer programmers to write such codes would be a hard and cumbersome task, which is why we have compilers to write such codes.

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language; which humans cannot understand. So, we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

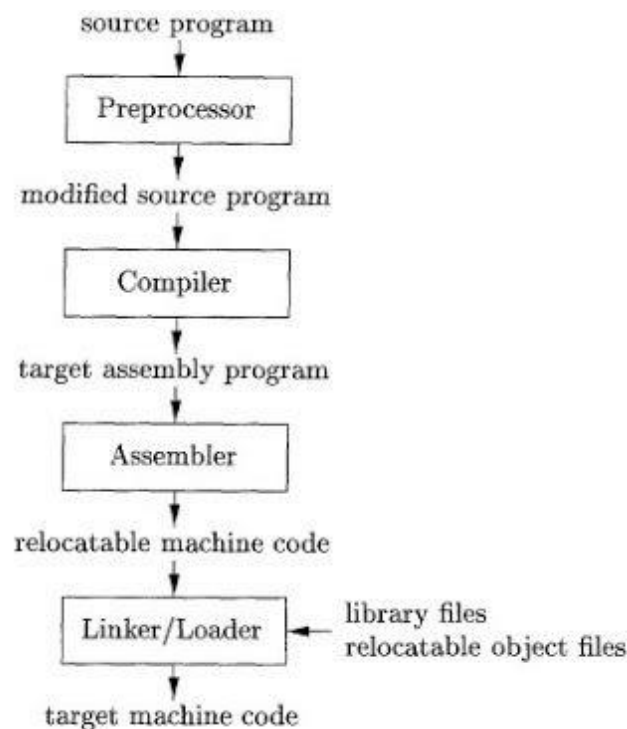


Fig: Language Processing System

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers. These are:

Preprocessor: A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter: An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it; whereas a compiler reads the whole program even if it encounters several errors.

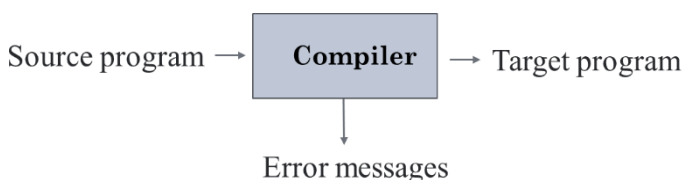
Assembler: An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker: Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader: Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

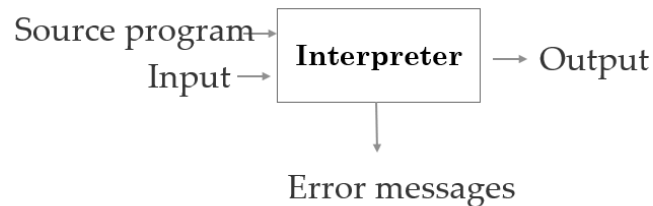
Language Processor: Compiler

A compiler is a program that takes a program written in a source language and translates it into an equivalent program in a target language.



Language Processor: Interpreter

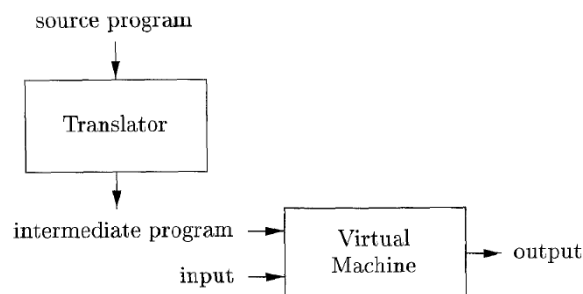
An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.



Hybrid Compiler:

Java language processors combine compilation and interpretation. A Java source program may first be compiled into an intermediate form called bytecodes.

The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.



Interpreter Vs. Compiler:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Compiler Architecture

A compiler can broadly be divided into two phases based on the way they compile.

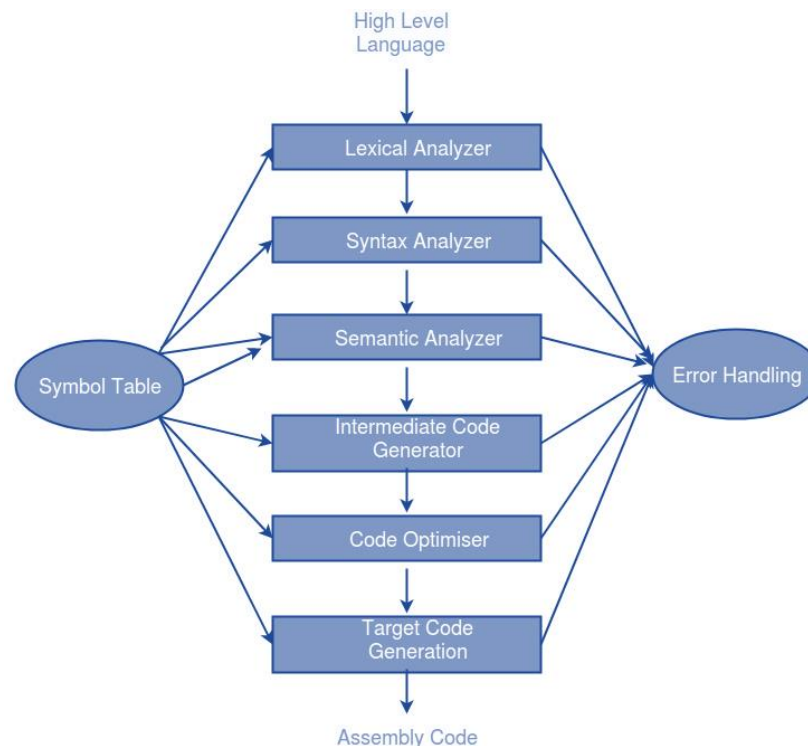
Analysis Phase: This phase of the compiler reads the source program, splits it into core parts, and then tests for lexical, grammatical, and syntax errors. The analysis phase is known as the front-end of the compiler. An intermediate representation of the source program and symbol table is generated by the analysis stage, which should be fed as input to the Synthesis stage.

Synthesis Phase: The synthesis stage is known as the compiler's back-end, with the help of intermediate source code representation and symbol table, this stage generates the target program. There can be several stages and passes of a compiler.

- **Pass:** A pass refers to the traversal of a compiler through the entire program.
- **Phase:** A compiler phase is a distinctive phase that takes input from the previous phase, methods, and outputs that can be used for the next phase as input. More than one step of a pass may have.

Compilation Steps/Phases:

- **Lexical Analysis:** Generates the “tokens” in the source program
- **Syntax Analysis:** Recognizes “sentences” in the program using the syntax of the language
- **Semantic Analysis:** Infers information about the program using the semantics of the language
- **Intermediate Code Generation:** Generates “abstract” code based on the syntactic structure of the program and the semantic information
- **Optimization:** Refines the generated code using a series of optimizing transformations
- **Final Code Generation:** Translates the abstract intermediate code into specific machine instructions



Lexical Analysis:

The first step of a compiler is lexical analysis. It takes from language preprocessors the modified source code that is written in the form of sentences. By deleting any white space or commentary in the source code, the lexical analyzer splits these sentences into a set of tokens. If a token is identified as invalid by the lexical analyzer, it produces an error. The lexical analyzer works with the syntax analyzer in close collaboration. It reads character streams from the source code, checks for legal tokens, and when necessary, passes the information to the syntax analyzer.

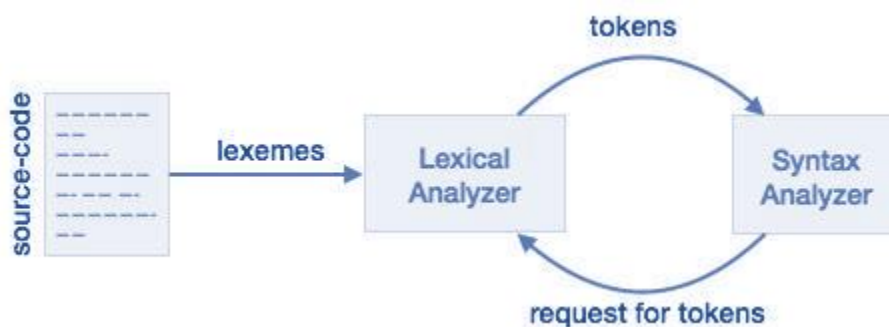


Fig: Working Principles of Lexical Analysis

Tokens:

It is said that lexemes are a series of (alphanumeric characters), also called tokens. For each lexeme to be identified as a valid token, there are some predefined rules. These rules, by means of a pattern, are described by grammar rules. A pattern describes what a token can be and by means of regular expressions, these patterns are described.

Keywords, constants, identifiers, sequences, numbers, operators, and punctuation symbols can be used as tokens in the programming language.

Example:

$x = y + 10$

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

Specifications of Tokens

Let us understand how the language theory considers the following terms:

Alphabets: Any finite set of symbols $\{0,1\}$ is a set of binary alphabets, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ is a set of Hexadecimal alphabets, $\{a-z, A-Z\}$ is a set of English language alphabets.

Strings: Any finite sequence of alphabets is called a string. Length of the string is the total number of alphabets in the string, e.g., the string S is "INDIA", the length of the string, S is 5 and is denoted by $|S|=5$. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ϵ (epsilon).

Special Symbols: A typical high-level language contains the following symbols: -

Symbols	Purpose
Addition(+), Subtraction(-), Modulo(%), Multiplication(*) and Division(/)	Arithmetic Operator
Comma(,), Semicolon(;), Dot(.), Arrow(->)	Punctuation
=, +=, /=, *=, -=	Assignment
==, !=, <, <=, >, >=	Comparison
#	Preprocessor
&	Location Specifier
&, &&, , , !	Logical
>>, >>>, <<, <<<	Shift Operator

The role of Lexical Analysis

1. It could be a separate pass, placing its output on an intermediate file from which the parser would then take its input.
2. The lexical analyzer and parser are together in the same pass; the lexical analyzer acts as a subroutine or co routine, which is called by the parser whenever it needs new token.
3. Eliminates the need for the intermediate file.
4. Returns a representation for the token it has found to the parser.

Syntax Analysis:

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not. The output of this phase is a parse tree.

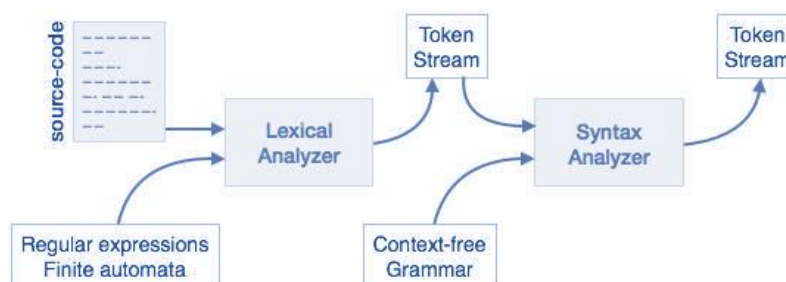


Fig: Working Principles of Syntax Analyzer

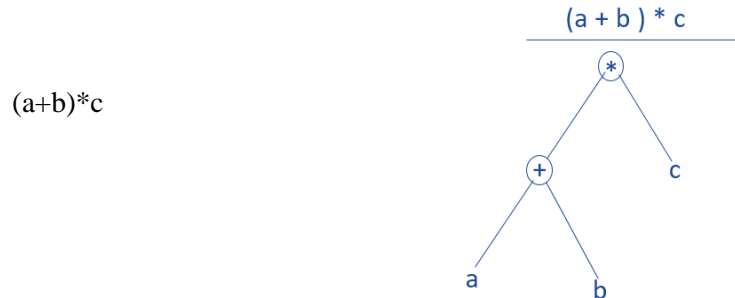
In this way, the parser accomplishes two tasks, parsing the code and looking for errors. Finally, a parse tree is generated as the output of this phase. Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies.

Example:

Any identifier/number is an expression

If x is an identifier and y+10 is an expression, then x= y+10 is a statement.

Consider parse tree for the following example



Limitations of Syntax Analyzers

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks:

- It cannot determine if a token is valid,
- It cannot determine if a token is declared before it is being used,
- It cannot determine if a token is initialized before it is being used,
- It cannot determine if an operation performed on a token type is valid or not.

Semantic Analysis:

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.

Example:

```
float x = 20.2;
```

```
float y = x*30;
```

Intermediate Code Generation:

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine.

Example:

For example,

```
total = count + rate * 5
```

Intermediate code with the help of address code method is:

```
t1 := int_to_float(5)
```

```
t2 := rate * t1
```

```
t3 := count + t2
```

```
total := t3
```

Code Optimization:

The next phase of is code optimization or Intermediate code. The term “code optimization” refers to techniques, a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

Example:

Consider the following code

```
a = intofloat(10)
b = c * a
d = e + b
f = d
Can become
b = c * 10.0
f = e+b
```

Code Generation:

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code.

Example:

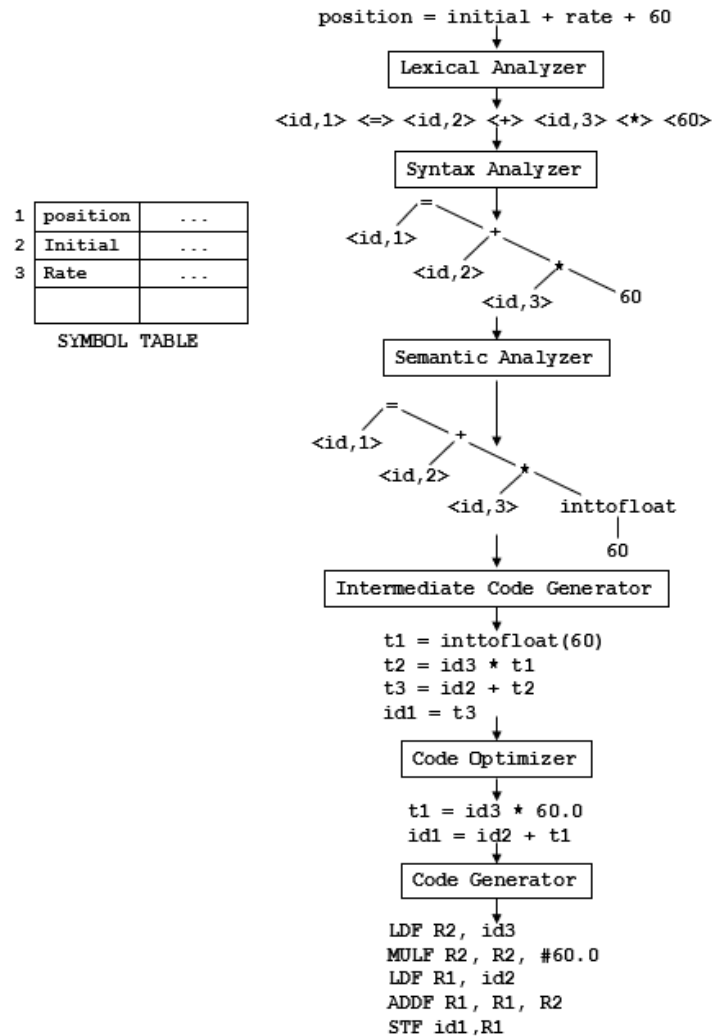
```
a = b + 60.0
Would be possibly translated to registers.
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2
```

Symbol Table Management:

A symbol table contains a record for each identifier with fields for the attributes of the identifier. This component makes it easier for the compiler to search the identifier record and retrieve it quickly. The symbol table also helps you for the scope management. The symbol table and error handler interact with all the phases and symbol table update correspondingly.

For example, suppose a source program contains the assignment statement

position = initial + rate * 60



Tokens, Patterns, and Lexemes:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token:

Token	lexeme	pattern
const	const	const
if	if	if
relation	<, <=, =, >, >=, >	< or <= or = or > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

Type of Errors:

Lexical: name of some identifier typed incorrectly

Syntactical: missing semicolon or unbalanced parenthesis

Semantical: incompatible value assignment

Logical: code not reachable, infinite loop

For example:

```
#include<stdio.h>
int main{
    innt a[2]={2,4,6}, b=1;
    sum = a[b]+b
    prntf("Rasalt is: %f, sum);
    return b;
}
```

Solution:

- i) **main** = Syntactical error
- ii) **innt** = Lexical error
- iii) **a[2]** = Semantic error
- iv) **sum** = Semantic error
- v) **;** = Syntactical error
- vi) **prntf()** = Lexical error
- vii) **%f** = Semantic error
- viii) **"** = Syntactical error

Errors Recovery Strategies:

There are 4 Errors Recovery Strategies.

They are:

- i) Panic mode
- ii) Statement mode
- iii) Error productions
- iv) Global correction

Panic Mode:

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement Mode:

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions:

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction:

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest errorfree statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Question:

1. What are the classification of compiler?

Answer: Compilers are classified as:

1. Single-pass
2. Multi-pass
3. Load-and-go
4. Debugging or optimizing

2. List the various compiler construction tool

Answer: The following is a list of some compiler construction tools:

1. Parser generators
2. Scanner generators
3. Syntax-directed translation engines
4. Automatic code generators
5. Data-flow engines

3. What tasks should be performed in semantic analysis?

Answer: The following tasks should be performed in semantic analysis:

1. Scope resolution
2. Type checking
3. Array-bound checking

4. Why we need an intermediate code?

Answer: 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. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. The second part of compiler, synthesis, is changed according to the target machine. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

5. In which way Intermediate codes can be represented?

Answer: There are 3 types of intermediate representations discussed below,

1. Post Fix
2. Syntax tree
3. 3-Address code, Quadruples and Triples.

6. Explain the principal sources of Optimization.

Answer: The Principal Sources of Optimization are given below:

- The code optimization techniques consist of detecting patterns in the program and replacing these patterns by equivalent but more efficient construct.
- Patterns may be local or global and replacement strategy may be machine dependent or machine dependent.

Exercise:

1. Why we Learn Compiler design?
2. What is Language Processing System?
3. What is Compiler and Interpreter?
4. Write down the difference between Compiler and Interpreter?
5. What is the architecture of Compiler?
6. Depict diagrammatically how a language is processed.
7. List and describe phase of a compiler.
8. Write down the role of Lexical Analysis.
9. Write down the limitations of Syntax Analyzers.

Chapter 2

Regular Expression

A Regular expression is a sequence of characters that defines a search pattern, particularly for use in patterns matching with strings, i.e., in searching for operations similar to "finding and substituting." For defining tokens, regular expression is a notation which usually used to define token language applications.

Sets of strings are the set of all integer constants or a set of all variable names Where a basic alphabet is taken from the individual letters. Such a set of strings is called a language. In whole numbers, the 0-9 and alphabet are the alphabet includes all letters and numbers (and even a few for variable names. Other characters, including underlining)

In the light of an alphabet, we define string sets with regular words, Compact and easy to use and understand algebraic notation for humans. The idea is to combine regular expressions describing simple string sets in order to construct regular expressions describing more complicated string sets.

When talking about regular expressions, we will use the letters (*r*, *s* and *t*) in italics to denote unspecified regular expressions. When letters stand for themselves (i.e., in regular expressions that describe strings that use these letters) we will use typewriter font, e.g., `a` or `b`. Hence, when we say, e.g., "The regular expression *s*" we mean the regular expression that describes a single one-letter string "*s*", but when we say "The regular expression *s*", we mean a regular expression of any form which we just happen to call *s*. We use the notation $L(s)$ to denote the language (i.e., set of strings) described by the regular expression *s*. For example, $L(a)$ is the set $\{“a”\}$.

Figure 2.1 shows the constructions used to build regular expressions and the languages they describe:

Regular Expression	Language (set of strings)	Informal description
a	$\{“a”\}$	The set consisting of the one-letter string “a”.
ϵ	$\{“”\}$	The set containing the empty string.
s t	$L(s) \cup L(t)$	Strings from both languages
st	$\{vw \mid v \in L(s), w \in L(t)\}$	Strings constructed by concatenating a string from the first language with a string from the second language. Note: In

		set-formulas, “ ” is not a part of a regular expression, but part of the setbuilder notation and reads as “where”.
s^*	$\{\epsilon\} \cup \{vw \mid v \in L(s), w \in L(s^*)\}$	Each string in the language is a concatenation of any number of strings in the language of s .

Figure 2.1: Regular expressions

- A single letter describes the language that has the one-letter string consisting of that letter as its only element.
- The symbol ϵ (the Greek letter epsilon) describes the language that consists solely of the empty string. Note that this is not the empty set of strings.
- $s|t$ (pronounced “s or t”) describes the union of the languages described by s and t .
- st (pronounced “s t”) describes the concatenation of the languages $L(s)$ and $L(t)$, i.e., the sets of strings obtained by taking a string from $L(s)$ and putting this in front of a string from $L(t)$. For example, if $L(s)$ is $\{“a”, “b”\}$ and $L(t)$ is $\{“c”, “d”\}$, then $L(st)$ is the set $\{“ac”, “ad”, “bc”, “bd”\}$.
- The language for s^* (pronounced “s star”) is described recursively: It consists of the empty string plus whatever can be obtained by concatenating a string from $L(s)$ to a string from $L(s^*)$. This is equivalent to saying that $L(s^*)$ consists of strings that can be obtained by concatenating zero or more (possibly different) strings from $L(s)$. If, for example, $L(s)$ is $\{“a”, “b”\}$ then $L(s^*)$ is $\{\epsilon, “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”, \dots\}$, i.e., any string (including the empty) that consists entirely of as and bs.

Some Exercises:

Define following operators over sets of strings:

- ❖ **Union: $L \cup U$**
 - $S = L \cup U = \{s \mid (s \in L) \vee (s \in U)\}$
- ❖ **Concatenation: LU or $L.U$**
 - $S = L . U = \{st \mid (s \in L) \wedge (t \in U)\}$
- ❖ **Kleene closure: L^* , set of all strings of letters, including ϵ ,**
 - $S = L^*$ denotes “zero or more concatenations of” L
- ❖ **Positive closure: L^+ .**
 - $S = L^+$ denotes “one or more concatenation of” L

Operations of Language

- ❖ Letters or alphabets and digits are the most important elements of language.
- ❖ Let L be the set of alphabets $\{A, B, \dots, Z, a, b, \dots, z\}$ and D be the set of digits $\{0, 1, \dots, 9\}$
- ❖ L could be in form of upper case and lower case
- ❖ Examples: $L \cup D$ is the set of letters and digits.
 - LD is the set of strings consisting of a letter followed by a digit.
- ❖ $LLLL = L^4$ is the set of all four-letter strings.
- ❖ L^* is the set of all strings of letters, including ϵ , the empty string
- ❖ $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
- ❖ D^* is the set of all strings of one or more digits.

Examples:

Let $L = \{a, b\}$

Some regular expressions:

- $a | b$
 - Denotes the set of $\{a, b\}$ having a or b .
- $(a|b)(a|b)$
 - Denotes $\{aa, ab, ba, bb\}$, the set of all strings of a 's and b 's of length two.
- a^*
 - Denotes the set of all strings of zero or more a 's, i. e., $\{\epsilon, a, aa, aaa, \dots\}$
- $(a|b)^*$ or $(a^*|b^*)^*$
 - Denotes the set of all strings containing zero or more instances of an a or b , that is, the set of all strings of a 's and b 's.
- $a | a^*b$
 - Denotes the set containing the string a and all strings consisting of zero or more a 's followed by a b
- $a(a | b)^*a$
 - String of a 's and b 's begin and end with a
- $(a | b)^* a(a | b) (a | b)$
 - String of a 's and b 's, with an a in the 3rd letter from the right.
- $(a | b)^*b (a | b)^* b (a | b)^*$
 - String of a 's and b 's that contain at least two b 's

Language to Regular Expressions

- “Set of all strings having at least one ab ”
 - $(ab)^+$
- “Set of all strings having even number of aa ”
 - $(aa)^*$
- “Set of all strings having odd number of bb ”
 - $b(bb)^*$
- “Set of all strings having zero or more instances of a or b starting with aa ”
 - $(aa)(a | b)^*$
- “Set of all strings having zero or more instances of a or b ending with bb ”
 - $(a | b)^* (bb)$
- String of a 's and b 's that contains odd number of b

- $a^*b(a^*ba^*b)^*a^*$
- String of a's and b's that contains just two or three b's
 - $a^*ba^*ba^*b^2a^*$
- All strings of a's and b's that do not contain the substring abb.
 - $b^*(a(\epsilon|b))^*$

Write regular definition for the following languages:

All string of lowercase letters that contain the five vowels in order.

Answer:

L $\rightarrow [b-d f-h j-n p-t v-z]$

String $\rightarrow L^*(a|A)^+ L^*(e|E)^+ L^*(i|I)^+ L^*(o|O)^+ L^*(u|U)^+ L^*$

Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes("")

Answer:

L $\rightarrow [a-zA-Z0-9]$

C $\rightarrow "*/"$

Commen $\rightarrow /* (L^*C^*)^* */$

Home Exercises:

1. What are the languages of this regular expressions?

- $(ab^*)^+c^?$
- $1^*0+1^?0$
- $(ab^*)bb(ab)^+$
- $(a+b)c^*$
- $aa(bc)^*cc$
- $abc(a+b+c)^+abc^*$

2. What are the regular expressions of these languages?

- all strings generated by $0^*(10^*)^*$. So they are equivalent
- "Set of all strings having zero or more instances of a or b starting with aa and ending with bb"
- All strings of a's and b's that do not contain the subsequence abb.
- All strings of a's and b's that contain at most two b's.
- All strings of a's and b's with an even number of a's.

3. Write a valid regular expression of an email address.

4. Write a valid regular expression of an URL address.

5. Write a valid expression of Facebook password.

6. Theory Questions:

- a. What is a regular expression?
- b. Why we use regular expression?
- c. What is the impotence of regular expression?
- d. Shows the constructions used to build regular expressions and the languages they describe.

Chapter 3

Context Free Grammars (CFG)

Like regular expressions, context-free grammars describe sets of strings, i.e., languages. Additionally, a context-free grammar also defines structure on the strings in the language it defines. A language is defined over some alphabet, for example the set of tokens produced by a lexer or the set of alphanumeric characters. The symbols in the alphabet are called terminals.

A context-free grammar recursively defines several sets of strings. Each set is denoted by a name, which is called a nonterminal. The set of nonterminal is disjoint from the set of terminals. One of the nonterminal are chosen to denote the language described by the grammar. This is called the start symbol of the grammar. The sets are described by a number of productions. Each production describes some of the possible strings that are contained in the set denoted by a nonterminal. A production has the form

$$N \rightarrow X_1 \dots X_n$$

When no confusion is likely, equate a nonterminal with the set of strings it denotes says that the set denoted by the nonterminal A contains the one-character string a.

Some examples:

$$A \rightarrow a$$

When several nonterminal are used, we must make it clear which of these is the start symbol. By convention (if nothing else is stated), the nonterminal on the left-hand side of the first production is the start symbol. As an example, the grammar has T as start symbol and denotes the set of strings that start with any number of as followed by a non-zero number of bs and then the same number of as with which it started.

$$T \rightarrow R$$

$$T \rightarrow aTa$$

$$R \rightarrow b$$

$$R \rightarrow bR$$

Sometimes, a shorthand notation is used where all the productions of the same nonterminal are combined to a single rule, using the alternative symbol (|) from regular expressions to separate the right-hand sides. In this notation, the above grammar would read. There are still four productions in the grammar, even though the arrow symbol \rightarrow is only used twice.

$$T \rightarrow R \mid aTa$$

$$R \rightarrow b \mid bR$$

A context-free grammar has four components: $G = (V, \Sigma, P, S)$

- non-terminals** (V): Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- terminal symbols** (Σ): A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- productions** (P): A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- start symbol** (S): One of the non-terminals is designated as the **start symbol** (S); from where the production begins.

Example of CFG:

$G = (V, \Sigma, P, S)$ Where:

$V = \{ Q, Z, N \}$

$\Sigma = \{ 0, 1 \}$

$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \epsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1 \}$

$S = \{ Q \}$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Notational Conventions:

To avoid always having to state that "these are the terminals," "these are the nonterminal," and so on, the following notational conventions for grammars will be used throughout the remainder of this book.

Terminals

a b c....

Nonterminals

A B C....

Grammar Symbols(Terminals or Nonterminals)

X Y Z U V W

String of Symbols <- A sequence of zero or more terminals and nonterminals

$\alpha \beta \gamma$

String of terminals <- Including ϵ

x y z u v w

Example:

$A \rightarrow \alpha B$ A rule whose righthand side ends with a nonterminal

$A \rightarrow x \alpha$ A rule whose righthand side begins with a string of terminals (call it "x")

Derivations:

- A. A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:
- B. Deciding the non-terminal which is to be replaced.
- C. Deciding the production rule, by which, the non-terminal will be replaced.
- D. To decide which non-terminal to be replaced with production rule, we can have two options.

CFG Terminology

Given

- G A grammar
- S The Start Symbol

Define

L(G) The language generated

$$L(G) = \{w \mid S \rightarrow^+ w\}$$

"Equivalence" of CFG's

If two CFG's generate the same language, we say they are **"equivalent"**.

$$G_1 \approx G_2 \text{ where } L(G_1) = L(G_2)$$

In making a derivation

Choose which nonterminal to expand

Choose which rule to apply

Example grammar:

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow id$$

Leftmost Derivation:	Rightmost Derivation:
$E \rightarrow E * E$	$E \rightarrow E + E$
$E \rightarrow E + E * E$	$E \rightarrow E + E * E$
$E \rightarrow id + E * E$	$E \rightarrow E + E * id$
$E \rightarrow id + id * E$	$E \rightarrow E + id * id$
$E \rightarrow id + id * id$	$E \rightarrow id + id * id$

Parse Tree:

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

Production rules:

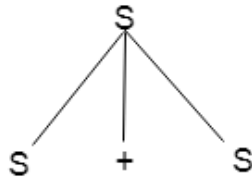
$T = T + T \mid T * T$

$T = a|b|c$

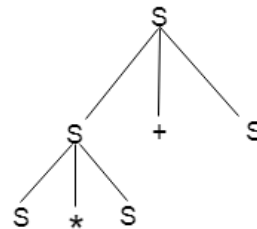
Input:

$a * b + c$

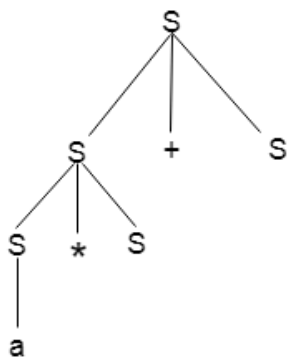
Step 1:



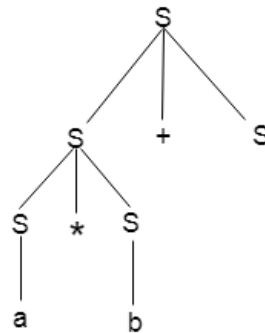
Step 2:



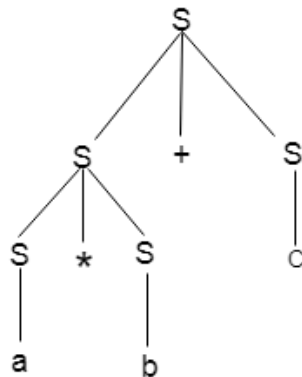
Step 3:



Step 4:



Step 5:



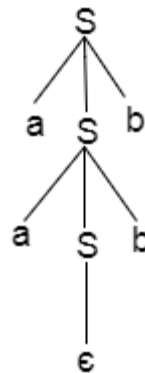
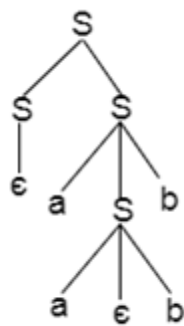
Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

- $S = aSb \mid SS$
- $S = \epsilon$

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

- $X \rightarrow Xa$

2. If the right associative operator (^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

- $X \rightarrow aX$

Example 1:

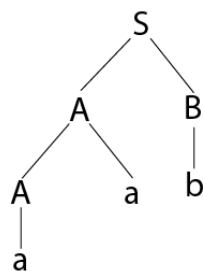
Consider a grammar G is given as follows:

- $S \rightarrow AB \mid aaB$
- $A \rightarrow a \mid Aa$
- $B \rightarrow b$

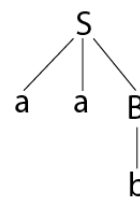
Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"



Parse tree 1



Parse tree 2

As there are two different parse trees for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

- $S \rightarrow AB$
- $A \rightarrow Aa \mid a$
- $B \rightarrow b$

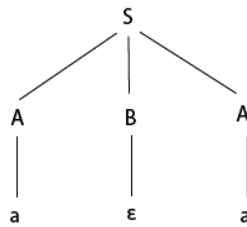
Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

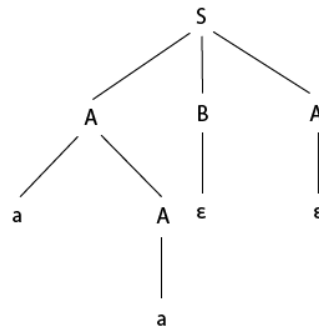
- $S \rightarrow ABA$
- $A \rightarrow aA \mid \epsilon$
- $B \rightarrow bB \mid \epsilon$

Solution:

The given grammar is ambiguous because we can derive two different parse trees for string aa.



Parse tree 1



Parse tree 2

The unambiguous grammar is:

- $S \rightarrow aXY \mid bYZ \mid \epsilon$
- $Z \rightarrow aZ \mid a$
- $X \rightarrow aXY \mid a \mid \epsilon$
- $Y \rightarrow bYZ \mid b \mid \epsilon$

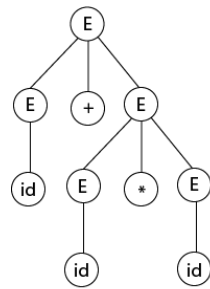
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

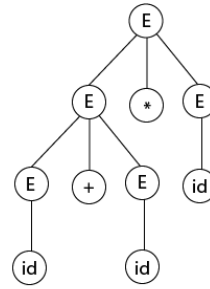
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow id$

Solution:

Let us derive the string "id + id * id"



Parse tree 1



Parse tree 2

As there are two different parse trees for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow id$

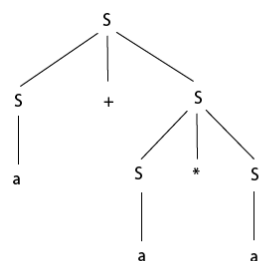
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

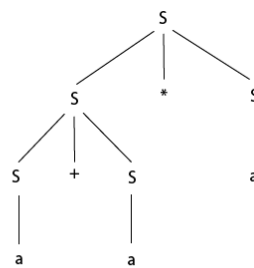
- $S \rightarrow S + S$
- $S \rightarrow S * S$
- $S \rightarrow S \wedge S$
- $S \rightarrow a$

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

- $S \rightarrow S + A \mid$
- $A \rightarrow A * B \mid B$
- $B \rightarrow C \wedge B \mid C$
- $C \rightarrow a$

Predictive Parsing

A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

Left Recursion:

Problem with Left Recursion:

If a left recursion is present in any grammar then, during parsing in the syntax analysis part of compilation there is a chance that the grammar will create infinite loop. This is because at every time of production of grammar S will produce another S without checking any condition.

Infinite Looping problem:

A grammar is left-recursive if it has a non-terminal A , such that there is a derivation:

$$A \Rightarrow A\alpha, \text{ for some } \alpha.$$

Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination.

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \dots \text{etc. } A \rightarrow A\alpha \mid \beta$$

So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

Algorithm to Remove Left Recursion with an example:

Suppose we have a grammar which contains left recursion:

$S \rightarrow S a / S b / c / d$

1. Check if the given grammar contains left recursion, if present then separate the production and start working on it.
In our example,
 $S \rightarrow S a / S b / c / d$
2. Introduce a new nonterminal and write it at the last of every terminal. We produce a new nonterminal S' and write new production as,
 $S \rightarrow cS' / dS'$
3. Write newly produced nonterminal in LHS and in RHS it can either produce or it can produce new production in which the terminals or non-terminals which followed the previous LHS will be replaced by new nonterminal at last.
 $S' \rightarrow ? / aS' / bS'$

So after conversion the new equivalent production is

$S \rightarrow cS' / dS'$
 $S' \rightarrow ? / aS' / bS'$

Left Factoring: Example

Do left factoring in the following grammar-

$S \rightarrow bSSa / bSSaSb / bSb / a$

Solution:

Step-01:

$S \rightarrow bSS' / a$

$S' \rightarrow SaaS / SaSb / b$

Again, this is a grammar with common prefixes.

Step-02:

$S \rightarrow bSS' / a$

$S' \rightarrow SaA / b$

$A \rightarrow aS / Sb$

This is a left factored grammar.

Questions:

1. Is the CFG is a Left Recursive? If so, remove the Left Recursive,
 $S \rightarrow Afc \mid be$
 $A \rightarrow Ac \mid Sdk \mid Be \mid ds \mid c$
 $B \rightarrow Ag \mid Sh \mid k$
 $C \rightarrow Bkma \mid AS \mid jb$
2. Consider the following grammar and eliminate left recursion-
 $A \rightarrow ABd \mid Aa \mid a$
 $B \rightarrow Be \mid b$
3. Consider the following grammar and eliminate left recursion-
 $E \rightarrow E + E \mid E \times E \mid a$
4. Consider the following grammar and eliminate left recursion-
 $E \rightarrow E + T \mid T$
 $T \rightarrow T \times F \mid F$
 $F \rightarrow id$
5. Consider the following grammar and eliminate left recursion-
 $S \rightarrow A$
 $A \rightarrow Ad \mid Ae \mid aB \mid ac$
 $B \rightarrow bBc \mid f$
6. Do left factoring in the following grammar-
 $S \rightarrow iEtS \mid iEtSeS \mid a$
 $E \rightarrow b$
7. Do left factoring in the following grammar-
 $A \rightarrow aAB \mid aBc \mid aAc$

Chapter 4

Non-Deterministic Finite Automata & Deterministic Finite Automata

In our quest to transform regular expressions into efficient programs, we use a stepping stone: Nondeterministic finite automata. By their nondeterministic nature, these are not quite as close to “real machines” as we would like, so we will later see how these can be transformed into deterministic finite automata, which are easily and efficiently executable on normal hardware.

A finite automaton is, in the abstract sense, a machine that has a finite number of states and a finite number of transitions between these. A transition between states is usually labelled by a character from the input alphabet, but we will also use transitions marked with ϵ , the so-called epsilon transitions.

A finite automaton can be used to decide if an input string is a member in some particular set of strings. To do this, we select one of the states of the automaton as the starting state. We start in this state and in each step, we can do one of the following:

- Follow an epsilon transition to another state, or
- Read a character from the input and follow a transition labelled by that character.

When all characters from the input are read, we see if the current state is marked as being accepting. If so, the string we have read from the input is in the language defined by the automaton.

We may have a choice of several actions at each step: We can choose between either an epsilon transition or a transition on an alphabet character, and if there are several transitions with the same symbol, we can choose between these. This makes the automaton nondeterministic, as the choice of action is not determined solely by looking at the current state and input. It may be that some choices lead to an accepting state while others do not. This does, however, not mean that the string is sometimes in the language and sometimes not: We will include a string in the language if it is possible to make a sequence of choices that makes the string lead to an accepting state.

This NFA recognizes the language described by the regular expression $a^*(a|b)$. As an example, the string aab is recognized by the following sequence of transitions:

from	to	by
1	2	ϵ
2	1	a
1	2	ϵ
2	1	a
1	3	b

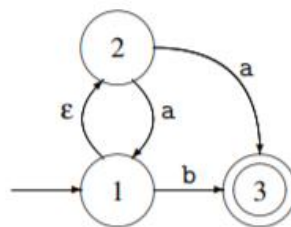


Figure 2.3: Example of an NFA

Automata:

An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

An automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where –

- Q is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the automaton.
- δ is the transition function.
- q_0 is the initial state from where any input is processed ($q_0 \in Q$)
- F is a set of final state/states of Q ($F \subseteq Q$)

#Related Terminologies

• Alphabet

Definition: An alphabet is any finite set of symbols.

Example: $\Sigma = \{a, b, c, d\}$ is an alphabet set where 'a', 'b', 'c', and 'd' are symbols.

• String

Definition: A string is a finite sequence of symbols taken from Σ .

Example: 'cabcad' is a valid string on the alphabet set $\Sigma = \{a, b, c, d\}$

• Length of a String

Definition: It is the number of symbols present in a string. (Denoted by $|S|$).

Examples: If $S = \text{'cabcad'}$, $|S| = 6$

If $|S| = 0$, it is called an empty string (Denoted by λ or ϵ)

• Kleene Star

Definition: The Kleene star, Σ^* , is a unary operator on a set of symbols or strings, Σ , that gives the infinite set of all possible strings of all possible lengths over Σ including λ .

Representation: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$ where Σ^p is the set of all possible strings of length p .

Example: If $\Sigma = \{a, b\}$, $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, \dots\}$

• Kleene Closure / Plus

Definition: The set Σ^+ is the infinite set of all possible strings of all possible lengths over Σ excluding λ .

Representation: $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

Example: If $\Sigma = \{a, b\}$,

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, \dots\}$$

• Language

Definition: A language is a subset of Σ^* for some alphabet Σ . It can be finite or infinite.

Example: If the language takes all possible strings of length 2 over $\Sigma = \{a, b\}$, then $L = \{ ab, aa, ba, bb \}$

Types of Finite Automaton:

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

NFA (Non-Deterministic finite automata)

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ϵ transition.

In the following image, we can see that from state q_0 for input a , there are two next states q_1 and q_2 , similarly, from q_0 for input b , the next states are q_0 and q_1 . Thus, it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deterministic finite automata.

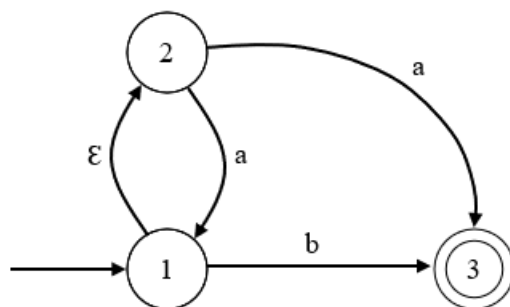


Figure: Example of a NFA

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

$$\delta: Q \times \Sigma \rightarrow 2Q$$

where,

- Q : finite set of states

- Σ : finite set of the input symbol
- q_0 : initial state
- F : final state
- δ : Transition function

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
4. The arc labeled with an input character show the transitions.
5. The initial state is marked with an arrow.
6. The final state is denoted by the double circle.

Example 1:

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

$q_0 = \{q_0\}$

$F = \{q_2\}$

Solution:

Transition diagram:

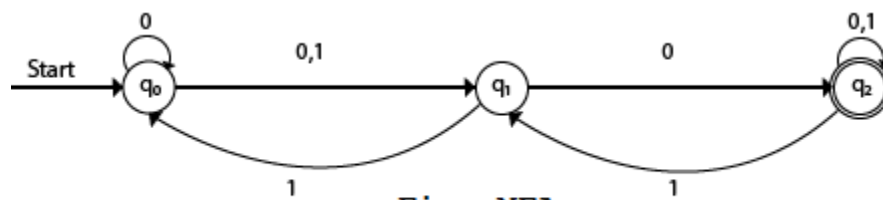


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0, q_1	q_1
q_1	q_2	q_0
$*q_2$	q_2	q_1, q_2

In the above diagram, we can see that when the current state is q_0 , on input 0, the next state will be q_0 or q_1 , and on 1 input the next state will be q_1 . When the current state is q_1 , on input 0 the next state will be q_2 and on 1 input, the next state will be q_0 . When the current state is q_2 , on 0 input the next state is q_2 , and on 1 input the next state will be q_1 or q_2 .

Example 2:

NFA with $\Sigma = \{0, 1\}$ accepts all strings with 01.

Solution:



Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→ q_0	q_1	ϵ
q_1	ϵ	q_2
* q_2	q_2	q_2

DFA (Deterministic finite automata)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q_0 for input a, there is only one path which is going to q_1 . Similarly, from q_0 , there is only one path for input b going to q_2 .

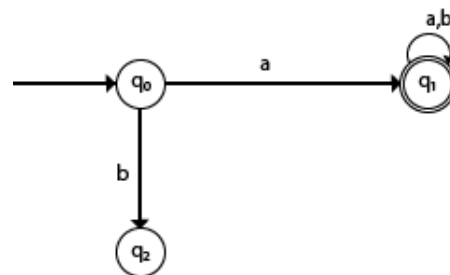


Fig:- DFA

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q : finite set of states
2. Σ : finite set of the input symbol
3. q_0 : initial state
4. F : final state
5. δ : Transition function

Transition function can be defined as:

- $\delta: Q \times \Sigma \rightarrow Q$

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

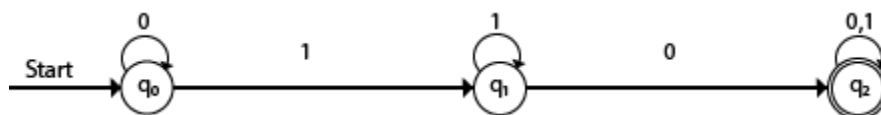
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $q_0 = \{q_0\}$
4. $F = \{q_2\}$

Solution:

Transition Diagram:



Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1

q1	q2	q1
*q2	q2	q2

DFA vs NFA

NFA refers to Nondeterministic Finite Automaton. DFA refers to Deterministic Finite Automaton. DFA can be best described and understood as one machine. NFA is like multiple small machines that are performing computational activities at the same time. All DFAs are derived from NFAs. The main difference between DFA and NFA, the two classes handling the transition functions of finite automata/ finite automaton theory, impact their behavior in many ways.

Basis of Difference	DFA	NFA
Reaction to symbols	For each symbolic representation of the alphabet, only a singular state transition can be attained in DFA.	No specifications are needed from the user with respect to how certain symbols impact the NFA.
Empty string transition	DFA is not capable of using an Empty String transition.	NFA can use an empty String transition.
Structure	DFA can be best described and understood as one machine.	NFA is like multiple small machines that are performing computational activities at the same time.
Rejection of string	DFA rejects the string in case it terminates in a state that is different from the accepting state.	NFA rejects the string in the event of all branches dying or refusing the string.
Backtracking	It is possible to use backtracking in DFA.	It is not possible to use backtracking at all times in the case of NFA.
Ease of construction	Given its complex nature, it is tougher to construct DFA.	NFA is more easily constructed in comparison to DFA.
Supremacy	All DFAs are derived from NFAs.	All NFAs are not DFAs.
Transition functions	The number related to the next state is one.	The number related to the next state/ states is either zero or one.

Complexities of time	The total time required for running any input string in DFA is less than what it is in NFA.	The total time required for running any input string in NFA is larger than that in comparison to DFA.
Full form	The full form of DFA is Deterministic Finite Automata.	The full form of NFA is Nondeterministic Finite Automata (NFA).
Space requirement	More space allocation needed.	Less space needed.
The setting of the next possible set	The next possible state is clearly set in DFA.	In NFA, every single pair of input symbols and states may contain many possible next states.

Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA which accepts the language $L(M)$. There should be equivalent DFA denoted by $M' = (Q', \Sigma', q'_0, \delta', F')$ such that $L(M) = L(M')$.

Steps for converting NFA to DFA:

Step 1: Initially $Q' = \phi$

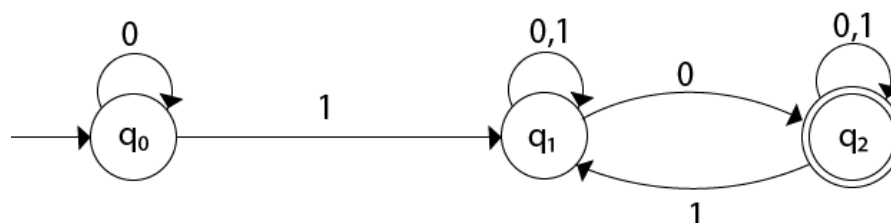
Step 2: Add q_0 of NFA to Q' . Then find the transitions from this start state.

Step 3: In Q' , find the possible set of states for each input symbol. If this set of states is not in Q' , then add it to Q' .

Step 4: In DFA, the final state will be all the states which contain F (final states of NFA)

Example 1:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

1. $\delta'([q0], 0) = [q0]$
2. $\delta'([q0], 1) = [q1]$

The δ' transition for state q1 is obtained as:

1. $\delta'([q1], 0) = [q1, q2]$ (new state generated)
2. $\delta'([q1], 1) = [q1]$

The δ' transition for state q2 is obtained as:

1. $\delta'([q2], 0) = [q2]$
2. $\delta'([q2], 1) = [q1, q2]$

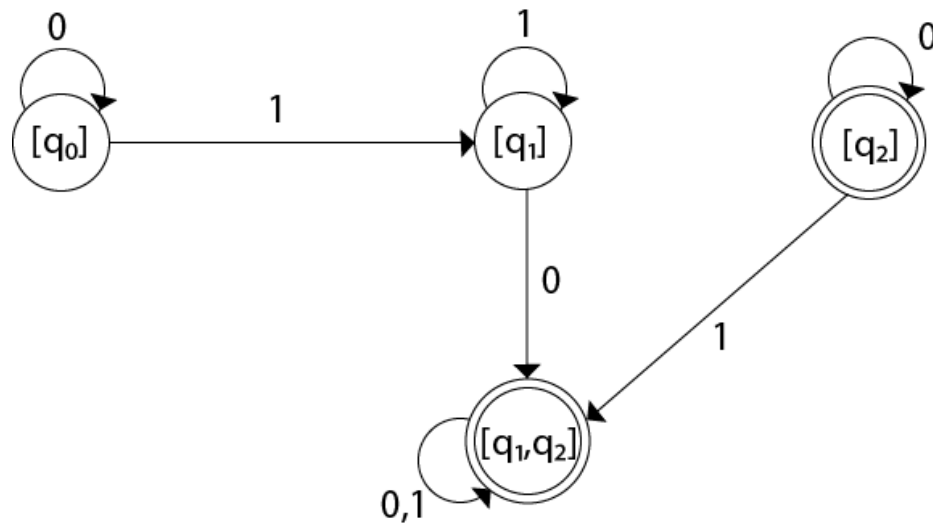
Now we will obtain δ' transition on [q1, q2].

1. $\delta'([q1, q2], 0) = \delta(q1, 0) \cup \delta(q2, 0)$
2. $= \{q1, q2\} \cup \{q2\}$
3. $= [q1, q2]$
4. $\delta'([q1, q2], 1) = \delta(q1, 1) \cup \delta(q2, 1)$
5. $= \{q1\} \cup \{q1, q2\}$
6. $= \{q1, q2\}$
7. $= [q1, q2]$

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

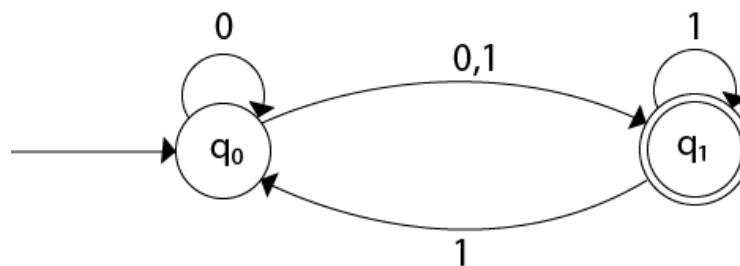
The Transition diagram will be:



The state q_2 can be eliminated because q_2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

Now we will obtain δ' transition for state q_0 .

1. $\delta'([q_0], 0) = \{q_0, q_1\}$
 $= [q_0, q_1]$ (new state generated)
2. $\delta'([q_0], 1) = \{q_1\} = [q_1]$

The δ' transition for state q_1 is obtained as:

1. $\delta'([q_1], 0) = \phi$
2. $\delta'([q_1], 1) = [q_0, q_1]$

Now we will obtain δ' transition on $[q_0, q_1]$.

1. $\delta'([q_0, q_1], 0) = \delta(q_0, 0) \cup \delta(q_1, 0)$
2. $= \{q_0, q_1\} \cup \phi$
3. $= \{q_0, q_1\}$
4. $= [q_0, q_1]$

Similarly,

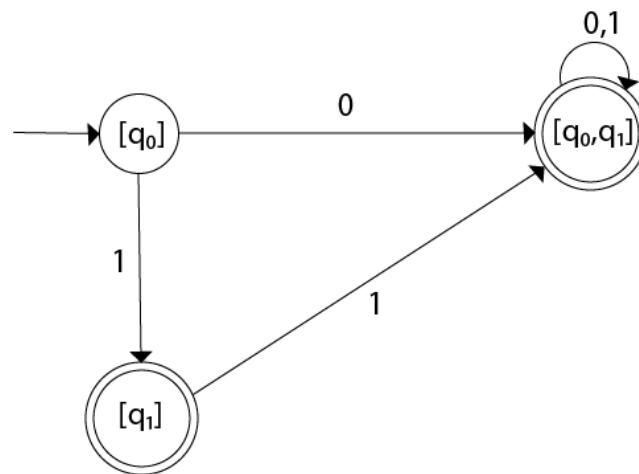
1. $\delta'([q_0, q_1], 1) = \delta(q_0, 1) \cup \delta(q_1, 1)$
2. $= \{q_1\} \cup \{q_0, q_1\}$
3. $= \{q_0, q_1\}$
4. $= [q_0, q_1]$

As in the given NFA, q_1 is a final state, then in DFA wherever, q_1 exists that state becomes a final state. Hence in the DFA, final states are $[q_1]$ and $[q_0, q_1]$. Therefore, set of final states $F = \{[q_1], [q_0, q_1]\}$.

The transition table for the constructed DFA will be:

State	0	1
$\rightarrow [q_0]$	$[q_0, q_1]$	$[q_1]$
$*[q_1]$	ϕ	$[q_0, q_1]$
$*[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_1]$

The Transition diagram will be:

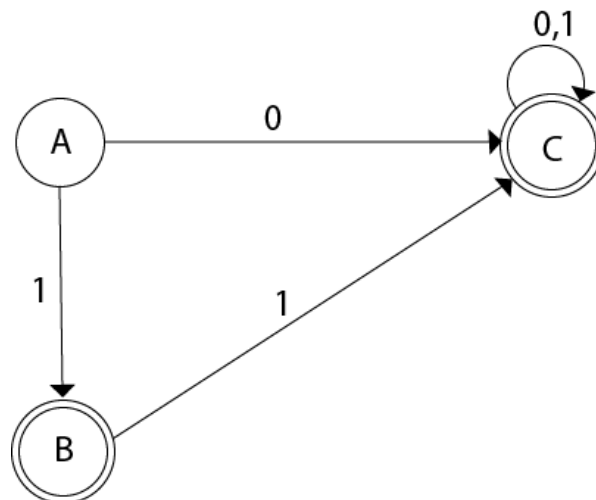


Even we can change the name of the states of DFA.

Suppose

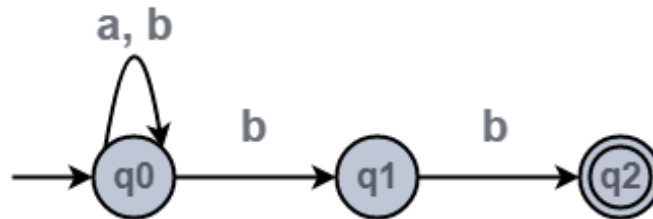
1. $A = [q_0]$
2. $B = [q_1]$
3. $C = [q_0, q_1]$

With these new names the DFA will be as follows:

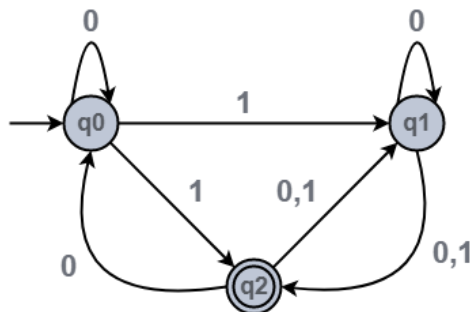


Questions:

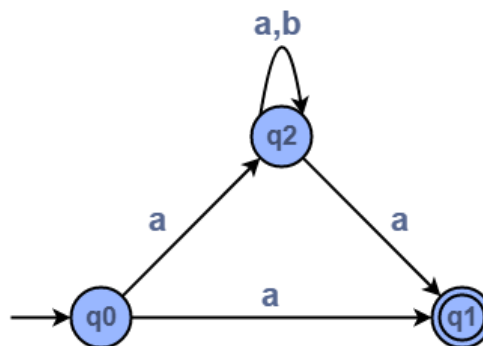
1. Draw a deterministic and non-deterministic finite automate which accept 00 and 11 at the end of a string containing 0, 1 in it, e.g., 01010100 but not 000111010.
2. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



3. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



4. Convert the following Non-Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)-



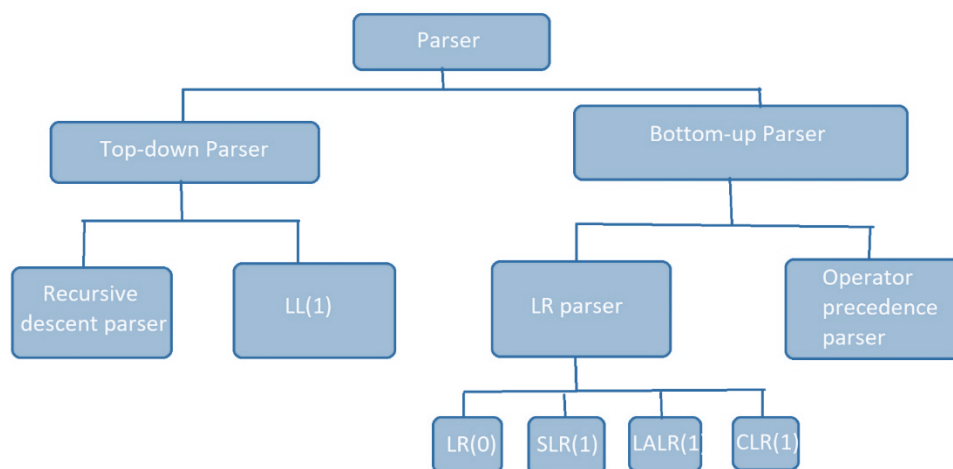
Chapter 5

Determination of FIRST and FOLLOW Function

Parsing

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.

Types of Parser: Parser is mainly classified into 2 categories: Top-down Parser, and Bottom-up Parser.



1. Top-down Parser: Top-down parser is the parser which generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into 2 types: Recursive descent parser, and Non-recursive descent parser.

- . Recursive descent parser: It is also known as Brute force parser or the with backtracking parser. It basically generates the parse tree by using brute force and backtracking.

- (ii). Non-recursive descent parser: It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses parsing table to generate the parse tree instead of backtracking.

2. Bottom-up Parser: Bottom-up Parser is the parser which generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses reverse of the right most derivation.

(i). LR parser:

LR parser is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar. It follows reverse of right most derivation.

LR parser is of 4 types:

- (a) LR(0)
- (b) SLR(1)
- (c) LALR(1)
- (d) CLR(1)

(ii). Operator precedence parser:

It generates the parse tree from given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear in the right-hand side of any production.

We have discussed about First and Follow and LL(1) in above discussion.

Example:

Find the first() follow() function and construct LL(1) Parse Table for the given grammar:

$P \rightarrow SQTRQ$

$Q \rightarrow SR \mid ce \mid \epsilon$

$R \rightarrow n \mid - \mid \epsilon$

$S \rightarrow T \mid hell$

$T \rightarrow fra \mid 0$

Solution:

Productions	First ()	Follow ()
$P \rightarrow SQTRQ$	{f, 0, h}	{ \$ }
$Q \rightarrow SR \mid ce \mid \epsilon$	{f, 0, h, c, ϵ }	{ \$, f, 0 }
$R \rightarrow n \mid - \mid \epsilon$	{n, -, ϵ }	{f, 0, h, c, \$ }
$S \rightarrow T \mid hell$	{f, 0, h}	{f, 0, h, c, n, -, \$ }
$T \rightarrow fra \mid 0$	{f, 0}	{n, -, f, 0, h, c, \$ }

Parse Table:

	c	n	-	h	f	0	\$
P				$P \rightarrow SQTRQ$	$P \rightarrow SQTRQ$	$P \rightarrow SQTRQ$	
Q	$Q \rightarrow ce$			$Q \rightarrow SR$	$Q \rightarrow SR \mid \epsilon$	$Q \rightarrow SR \mid \epsilon$	$Q \rightarrow \epsilon$
R	$R \rightarrow c$	$R \rightarrow n$	$R \rightarrow -$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$	$R \rightarrow \epsilon$
S					$S \rightarrow T$	$S \rightarrow T$	
T					$T \rightarrow fra$	$T \rightarrow 0$	

Exercise:

1. Find the first() follow() function and construct LL(1) Parse Table for:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

2. Find the first() follow() function and construct LL(1) Parse Table for:

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid$

$D \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$

3. Find the first() follow() function and construct LL(1) Parse Table for:

$S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

Chapter 6

LR0 Parser and Canonical Table

LR0 Parser:

LR parsers are a type of bottom-up parser that analyses deterministic context-free languages in linear time. There are several variants of LR parsers: SLR parsers, LALR parsers, Canonical LR(1) parsers, Minimal LR(1) parsers, GLR parsers. LR parsers can be generated by a parser generator from a formal grammar defining the syntax of the language to be parsed. They are widely used for the processing of computer languages.

An LR parser (Left-to-right, Rightmost derivation in reverse) reads input text from left to right without backing up (this is true for most parsers), and produces a rightmost derivation in reverse: it does a bottom-up parse – not a top-down LL parse or ad-hoc parse. The name LR is often followed by a numeric qualifier, as in LR(1) or sometimes LR(k). To avoid backtracking or guessing, the LR parser is allowed to peek ahead at k lookahead input symbols before deciding how to parse earlier symbols. Typically, k is 1 and is not mentioned. The name LR is often preceded by other qualifiers, as in SLR and LALR.

LR parsers are deterministic; they produce a single correct parse without guesswork or backtracking, in linear time. This is ideal for computer languages, but LR parsers are not suited for human languages which need more flexible but inevitably slower methods.

Before making LR0 parse table at first, we have to make canonical table.

Advantages of LR0 Parser:

- A large class of context-free grammars can be managed by LR parsers.
- The LR parsing method is a most general non-back tracking shift-reduce parsing method.
- The syntax errors can be identified as soon as an LR parser occurs.
- More languages than LL grammar can be represented in LR grammars.

Canonical Table:

A canonical object means an object with a single reference pointed to it, with no copies holding the same state possible.

The activity of replacing multiple copies of an object with just a few objects is often referred to as canonicalizing objects. The whole completed object is called Canonical Table.

Let's see few examples and solutions to make LR0 parser and Canonical Table from given expressions.

Example:

Construct LR(0) table and Canonical Table for the given grammar.

Given grammar:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

Add Augment Production and insert ' \cdot ' symbol at the first position for every production in G

$S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

I0 State:

Add Augment production to the I0 State and Compute the Closure

$I0 = \text{Closure}(S' \rightarrow \cdot S)$

Add all productions starting with S in to I0 State because " \cdot " is followed by the non-terminal. So, the I0 State becomes

$I0 = S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

Add all productions starting with "A" in modified I0 State because " \cdot " is followed by the non-terminal. So, the I0 State becomes.

$I0 = S' \rightarrow \cdot S$

$S \rightarrow \cdot AA$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

$I1 = \text{Go to}(I0, S) = \text{closure}(S' \rightarrow S \cdot) = S' \rightarrow S \cdot$

Here, the Production is reduced so close the State.

$I1 = S' \rightarrow S \cdot$

$I2 = \text{Go to}(I0, A) = \text{closure}(S \rightarrow A \cdot A)$

Add all productions starting with A in to I2 State because " \cdot " is followed by the non-terminal. So, the I2 State becomes

$I2 = S \rightarrow A \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

Go to $(I2, a) = \text{Closure}(A \rightarrow a \cdot A) = (\text{same as } I3)$

Go to $(I2, b) = \text{Closure}(A \rightarrow b \cdot) = (\text{same as } I4)$

$I3 = \text{Go to}(I0, a) = \text{Closure}(A \rightarrow a \cdot A)$

Add productions starting with A in I3.

$A \rightarrow a \cdot A$

$A \rightarrow \cdot aA$

$A \rightarrow \cdot b$

Go to (I3, a) = Closure ($A \rightarrow a \bullet A$) = (same as I3)

Go to (I3, b) = Closure ($A \rightarrow b \bullet$) = (same as I4)

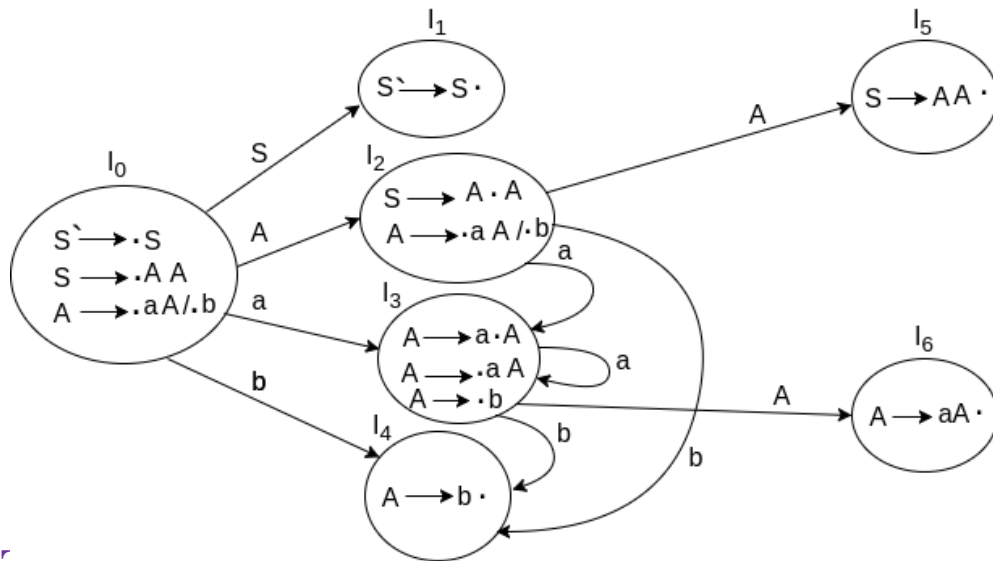
I4 = Go to (I0, b) = closure ($A \rightarrow b \bullet$) = $A \rightarrow b \bullet$

I5 = Go to (I2, A) = Closure ($S \rightarrow AA \bullet$) = $SA \rightarrow A \bullet$

I6 = Go to (I3, A) = Closure ($A \rightarrow aA \bullet$) = $A \rightarrow aA \bullet$

Canonical Table:

Let's construct the Canonical Table:



LR(0) :

- If a state is going to some other state on a terminal then it corresponds to a shift move.
- If a state is going to some other state on a variable then it corresponds to go to move.
-
- If a state contains the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	b	\$	A	S
I ₀	S3	S4		2	1
I ₁			accept		
I ₂	S3	S4		5	
I ₃	S3	S4		6	
I ₄	r3	r3	r3		
I ₅	r1	r1	r1		
I ₆	r2	r2	r2		

Explanation of LR0 Table:

I0 on S is going to I1 so write it as 1.

I0 on A is going to I2 so write it as 2.

I2 on A is going to I5 so write it as 5.

I3 on A is going to I6 so write it as 6.

I0, I2 and I3 on a are going to I3 so write it as S3 which means that shift 3.

I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.

I4, I5 and I6 all states contain the final item because they contain • in the right most end. So, rate the production as production number.

Example:

Construct LR(0) table and Canonical Table for the given grammar.

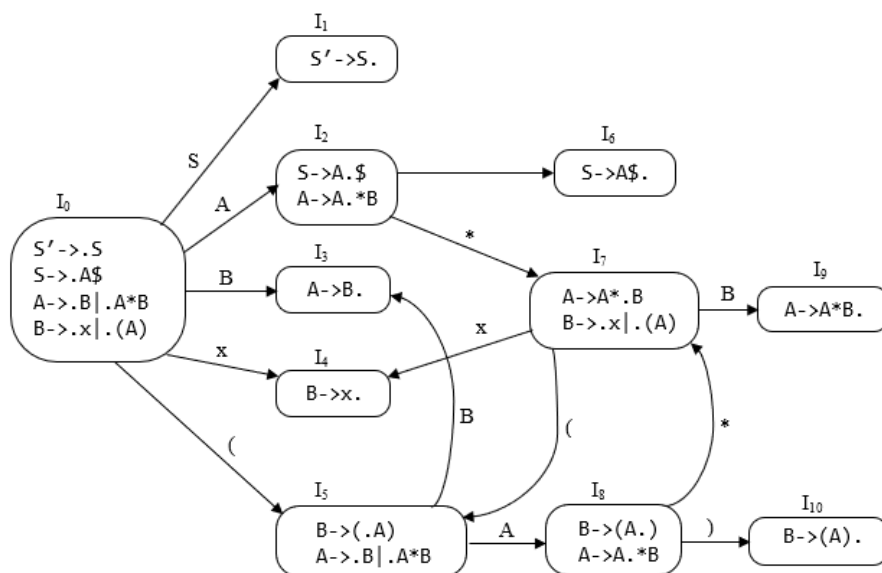
Given grammar:

$S \rightarrow A\$$

$A \rightarrow B|A*B$

$B \rightarrow x|(A)$

LR(0) Parser:



Canonical Table:

States	Action						Go to		
	x	*	()	\$	\$'	S	A	B
I ₀	S ₄		S ₅				1	2	3
I ₁						accept			
I ₂		S ₇			S ₆				
I ₃	r ₂	r ₂	r ₂	r ₂	r ₂	r ₂			
I ₄	r ₄	r ₄	r ₄	r ₄	r ₄	r ₄			
I ₅								8	3
I ₆	r ₁	r ₁	r ₁	r ₁	r ₁	r ₁			
I ₇	S ₄		S ₅						9
I ₈		S ₇		S ₁₀					
I ₉	r ₃	r ₃	r ₃	r ₃	r ₃	r ₃			
I ₁₀	r ₅	r ₅	r ₅	r ₅	r ₅	r ₅			

Exercise:

Question 1: Find the LR (0) Parser and Canonical Table:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Question 2: Find the LR (0) Parser and Canonical Table:

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

Question 3: Find the LR (0) Parser and Canonical Table:

$S \rightarrow aTRe$

$T \rightarrow Tbc \mid b$

$R \rightarrow d$

Question 4: Find the LR (0) Parser and Canonical Table:

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid c$

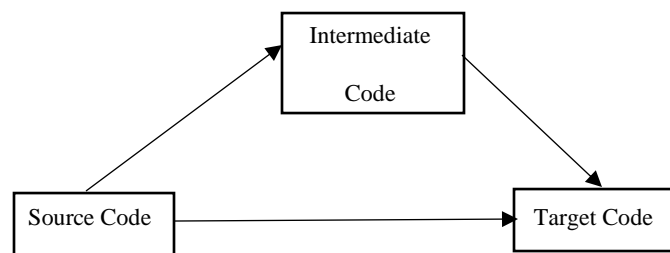
Questions:

1. What is LR (0) parser?
2. Why we need LR (0) parser?
3. What is Canonical Table?

Chapter - 7

Intermediate Code Generation

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language. A compiler can generate a middle-level language code, known as intermediate code or intermediate text, during the translation of a source program into an object code for a target machine. Between the source language code and the object code lies the ambiguity of this code. Intermediate code can be represented as a postfix notation, a syntax tree, a directed acyclic graph (DAG), a three-address code, a quadruple, a triple.



Importance of Intermediate Code Generation

- 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.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Advantages of using an intermediate code for the computer

- Portability would be improved because of the machine's independent intermediate code. For example, consider that if a programmer converts the source language into its target machine language without the option of producing intermediate code, a complete native compiler is needed for each new machine. Since there were, clearly, some changes according to the system requirements in the compiler itself.
- Retargeting is facilitated.
- To boost the performance of source code by improving the intermediate code, it is simpler to apply source code modification.

Intermediate representation

Intermediate code can be represented in two ways:

1. High Level intermediate code: High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.
2. Low Level intermediate code: Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

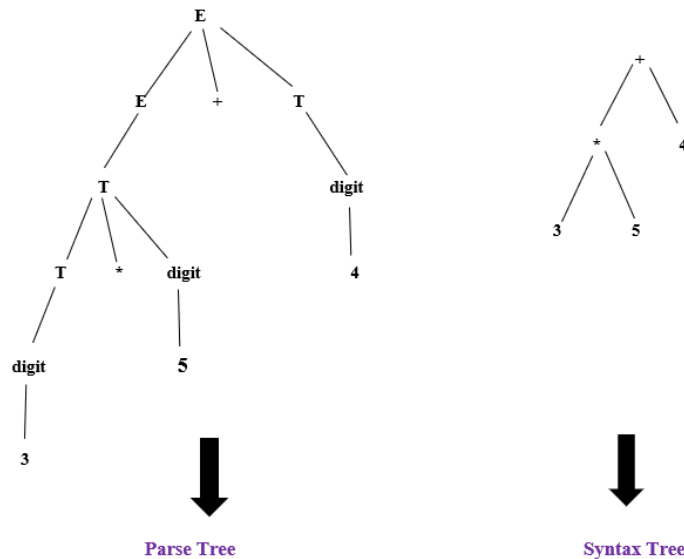
Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Questions:

1. Define Intermediate Code Generation.
2. Why we need ICG?
3. Write down the advantages of ICG.
4. In how many ways ICG can be presented?

Syntax Tree

In compilers, Syntax Trees are data structures commonly used to describe the program code structure. Typically, a Syntax Tree is the product of a compiler's syntax analysis level. It frequently represents the program through a variety of steps, which the compiler needs, and has a direct influence on the compiler's final performance. Syntax trees are abstract or compact representation of parse trees. They are also called as Abstract Syntax Trees. Example:



Syntax trees are called as Abstract Syntax Trees:

Because:

- They are abstract representation of the parse trees.
- They do not provide every characteristic information from the real syntax.
- For example- no rule nodes, no parenthesis etc.

Characteristics of Syntax Trees:

Syntax Trees have many characteristics that help the compilation process's further steps:

- For any element it contains, a Syntax Tree can be edited and enhanced with information such as properties and annotations. With a program's source code, such editing and annotation is difficult, because it would mean modifying it.
- A Syntax Tree does not have necessary punctuation and delimiters as compared to the source code (braces, semicolons, parentheses, etc.).

- Due to the consecutive stages of review by the compiler, a Syntax Tree normally includes extra details about the program. For example, it can store each element's location in the source code, enabling the compiler to print error messages that are useful.

Design of a Syntax Tree:

The design of a Syntax Tree is also closely related to a compiler's design and its anticipated features.

The core specifications include the following:

- Variable forms, as well as the position of each declaration in the source code, must be maintained.
- It is important to specifically represent and well describe the order of executable statements.
- It is important to store and correctly classify the left and right components of binary operations.
- For assignment statements, identifiers and their assigned values must be stored.

Usage of Syntax Tree:

During semantic analysis, the Syntax Tree is used intensively, where the programmer tests for the correct use of the program elements and the language. During semantic analysis, the compiler also produces symbol tables based upon the Syntax Tree. A full tree traversal enables the correctness of the program to be checked.

The Syntax Tree serves as the basis for code generation after checking correctness. The Syntax Tree for code generation is also used to produce an intermediate representation (IR), also called an intermediate language.

Differences between parse tree and syntax tree:

The difference between parse tree and syntax tree is given below:

Parse Tree	Syntax Tree
Parse tree is a graphical representation of the replacement process in a derivation.	Syntax tree is the compact form of a parse tree.
Each interior node represents a grammar rule. Each leaf node represents a terminal.	Each interior node represents an operator. Each leaf node represents an operand.
Parse trees provide every characteristic information from the real syntax.	Syntax trees do not provide every characteristic information from the real syntax.

Parse trees are comparatively less dense than syntax trees.

Syntax trees are comparatively denser than parse trees.

Example:

Q1. Construct a syntax tree for the following arithmetic expression:

$(a + b) * (c - d) + ((e / f) * (a + b))$

Solution:

Step-01:

We convert the given arithmetic expression into a postfix expression as:

$(a + b) * (c - d) + ((e / f) * (a + b))$

ab+ * (c - d) + ((e / f) * (a + b))

ab+ * cd- + ((e / f) * (a + b))

ab+ * cd- + (ef/ * (a + b))

ab+ * cd- + (ef/ * ab+)

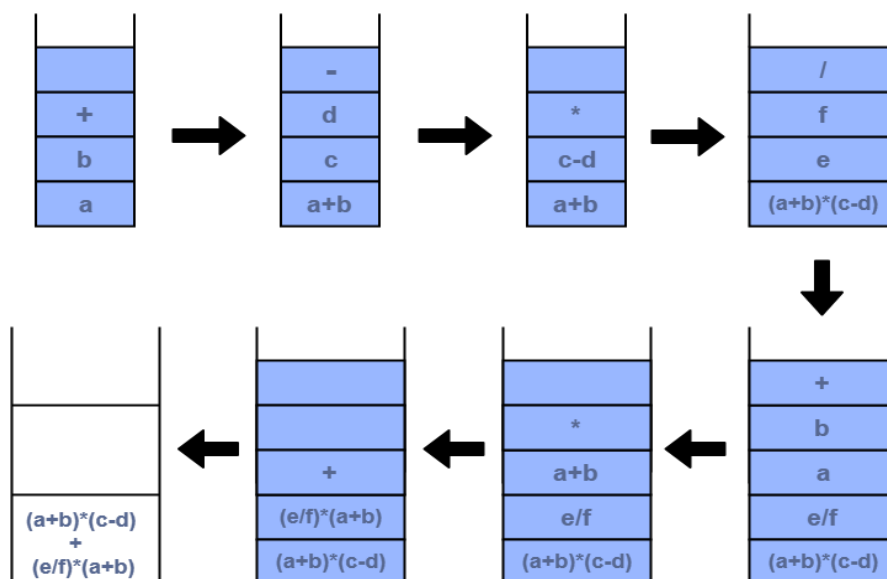
ab+ * cd- + ef/ab+*

ab+cd-* + ef/ab+*

ab+cd-*ef/ab+*+

Step-02:

Then we draw a syntax tree for the above postfix expression:



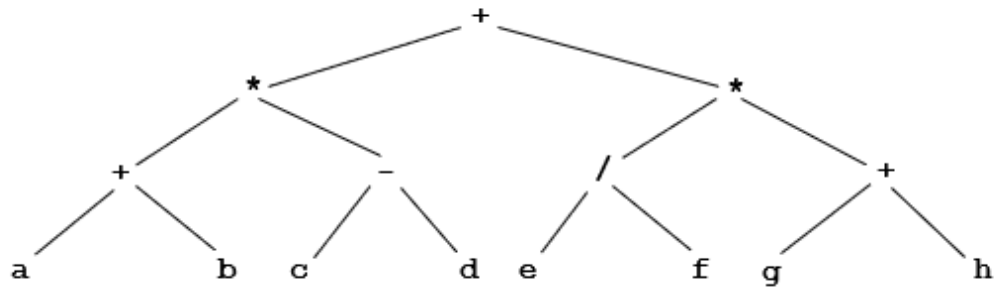


Fig: Syntax Tree

Q2. Construct a syntax tree for the following arithmetic expression:

$(a + a) + b * c + (b * c + c) + (d + d + d + d)$

Solution:

Step 1:

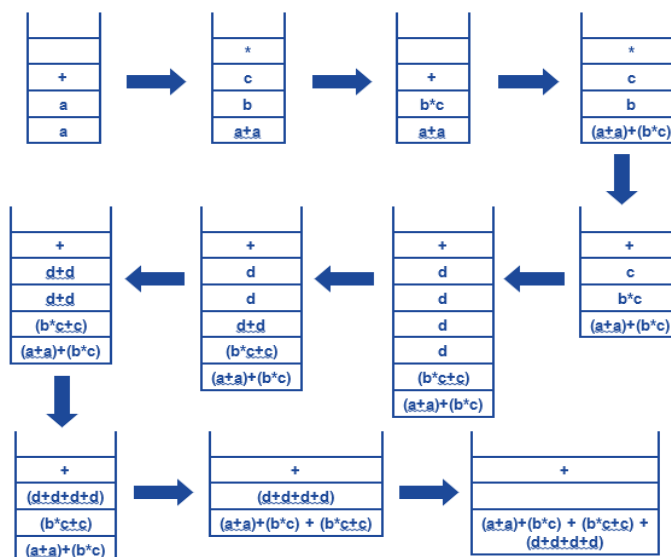
We convert the given arithmetic expression into a postfix expression as:

$(a + a) + b * c + (b * c + c) + (d + d + d + d)$

Postfix: aa+bc*+bc*c+d+d+d+d+++++

Step 2:

Then we draw a syntax tree for the above postfix expression:



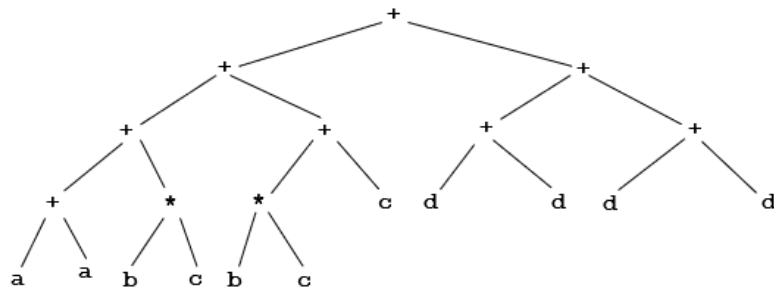


Fig: Syntax Tree

Exercises:

Question 1: Construct a syntax tree for the following arithmetic expression:

$$b * - c + b * - c$$

Question 2: Construct a syntax tree for the following arithmetic expression:

$$a + a * (b - c) + (b - c) * d$$

Question 3: Construct a syntax tree for the following arithmetic expression:

$$(a + b) * (c + d) + (a + b + c)$$

Questions:

1. What is a Syntax Tree? Explain with figures.
2. Why Syntax Tree is called Abstract Syntax Tree?
3. What is the difference between Parse tree and Syntax tree?
4. Write down the properties of Syntax Tree.

Chapter - 8

Directed Acyclic Graph (DAG)

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks.

Optimization of Basic Blocks:

- A DAG is constructed for optimizing the basic block.
- A DAG is usually constructed using Three Address Code.
- Transformations such as dead code elimination and common sub expression elimination are then applied.

Properties:

- Reachability relation forms a partial order in DAGs.
- Both transitive closure & transitive reduction is uniquely defined for DAGs.
- Topological Orderings are defined for DAGs.

Applications:

DAGs are used for the following purposes-

- To determine the expressions which have been computed more than once (called common sub-expressions).
- To determine the names whose computation has been done outside the block but used inside the block.
- To determine the statements of the block whose computed value can be made available outside the block.
- To simplify the list of Quadruples by not executing the assignment instructions $x:=y$ unless they are necessary and eliminating the common sub-expressions.

Rules for the construction of the DAG:

Rule-01:

In a DAG-

- Interior nodes always represent the operators.
- Exterior nodes (leaf nodes) always represent the names, identifiers or constants.

Rule-02:

While constructing a DAG,

- A check is made to find if there exists any node with the same value.
- A new node is created only when there does not exist any node with the same value.
- This action helps in detecting the common sub-expressions and avoiding the re-computation of the same.

Rule-03:

The assignment instructions of the form $x:=y$ are not performed unless they are necessary.

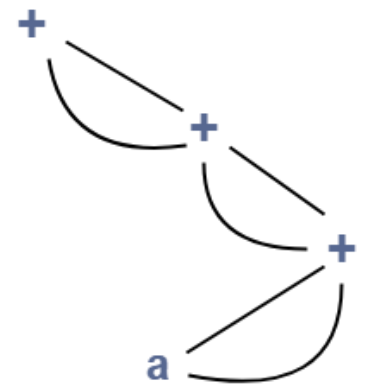
Example:

Consider the following expression and construct a DAG for it-

$(((a + a) + (a + a)) + ((a + a) + (a + a)))$

Solution:

Directed Acyclic Graph for the given expression is-



Directed Acyclic Graph

Exercise:

1. Consider the following expression and construct a DAG for:
$$(a + b) \times (a + b + c)$$
2. Consider the following block and construct a DAG for-
 - i. $a = b \times c$
 - ii. $d = b$
 - iii. $e = d \times c$
 - iv. $b = e$
 - v. $f = b + c$
 - vi. $g = f + d$
3. Consider the following expression and construct a DAG for it-
$$(a + b) \times (a + b + c)$$
4. Consider the following basic block and draw a directed acyclic graph.

B10:

```
S1 = 4 x I
S2 = addr(A) - 4
S3 = S2[S1]
S4 = 4 x I
S5 = addr(B) - 4
S6 = S5[S4]
S7 = S3 x S6
S8 = PROD + S7
PROD = S8
S9 = I + 1
I = S9
If I <= 20 goto L10
```

5. Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

```
for i=1 ... 10 do
  for j=1 ... 10 do
    a[i,j] = 0.0;

for i=1 ... 10 do
  a[i,i] = 1.0;
```


Chapter - 9

Three Address Code

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

The characteristics of Three Address Code are given below:

- They are generated by the compiler for implementing Code Optimization.
- They use maximum three addresses to represent any statement.
- They are implemented as a record with the address fields.

General Form:

In general, Three Address instructions are represented as-

a = b op c

Here,

- a, b and c are the operands.
- Operands may be constants, names, or compiler generated temporaries.
- op represents the operator.

Examples:

Examples of Three Address instructions are-

- $a = b + c$
- $c = a \times b$

Common Three Address Instruction Forms:

The common forms of Three Address instructions are-

1. Assignment Statement:

$x = y \text{ op } z$ and $x = \text{op } y$

Here,

- x , y and z are the operands.
- op represents the operator.

It assigns the result obtained after solving the right-side expression of the assignment operator to the left side operand.

2. Copy Statement:

$x = y$

Here,

- x and y are the operands.
- $=$ is an assignment operator.

It copies and assigns the value of operand y to operand x .

3. Conditional Jump:

If $x \text{ relop } y$ goto X

Here,

- x & y are the operands.
- X is the tag or label of the target statement.
- relop is a relational operator.

If the condition “ $x \text{ relop } y$ ” gets satisfied, then-

- The control is sent directly to the location specified by label X .
- All the statements in between are skipped.

If the condition “ $x \text{ relop } y$ ” fails, then-

- The control is not sent to the location specified by label X .

- The next statement appearing in the usual sequence is executed.

4. Unconditional Jump:

goto X

Here, X is the tag or label of the target statement.

On executing the statement,

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

5. Procedure Call:

param x call p return y

Here, p is a function which takes x as a parameter and returns y.

Example:

Write three address code for $a := (-c * b) + (-c * d)$

Three-address code is as follows:

```
t1 := -c
t2 := b*t1
t3 := -c
t4 := d * t3
t5 := t2 + t4
a := t5
```

Implementation of Three Address Code:

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple:

It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage:

- Easy to rearrange code for global optimization.

- One can quickly access value of temporary variables using symbol table.

Disadvantage:

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

2. Triples:

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

Disadvantage:

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

3. Indirect Triples:

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example:

Write quadruple, triples and indirect triples for following expression:

$$(x + y) * (y + z) + (x + y + z)$$

Solution:

The three-address code is:

$$t1 = x + y$$

$$t2 = y + z$$

$$t3 = t1 * t2$$

$$t4 = t1 + z$$

$$t5 = t3 + t4$$

Quadruple:

#	Op	Arg1	Arg2	Arg3
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Triples:

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Indirect Triples:

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Exercise:

1. Write Three Address Code for $a = b + c + d$
2. Write Three Address Code for $-(a \times b) + (c + d) - (a + b + c + d)$
3. Write Three Address Code for If $A < B$ then 1 else 0
4. Write Three Address Code for If $A < B$ and $C < D$ then $t = 1$ else $t = 0$
5. Write quadruple, triples and indirect triples for following expression:

$$a = b * -c + b * -c$$

6. Translate the following expression to quadruple, triple and indirect triple-

$$a + b \times c / e \uparrow f + b \times c$$

7. Translate the following expression to quadruple, triple and indirect triple-

$$a = b \times -c + b \times -c$$

Chapter - 10

Basic Block and Flow Diagram

Basic Block:

This section provides a graph representation of intermediate code that is useful even if the graph is not constructed specifically to address code generation by an algorithm of code-generation. Context benefits from code generation.

Basic Block is a straight-line code sequence which has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements which always executes one after other, in a sequence.

The first task is to partition a sequence of three-address code into basic blocks. A new basic block is begun with the first instruction and instructions are added until a jump or a label is met. In the absence of jump control moves further consecutively from one instruction to another.

Algorithm:

Partitioning three-address code into basic blocks.

Input:

A sequence of three address instructions.

Output:

A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

Process:

Instructions from intermediate code which are leaders are determined. Following are the rules used for finding leader:

The first three-address instruction of the intermediate code is a leader.

Instructions which are targets of unconditional or conditional jump/goto statements are leaders.

Instructions which immediately follows unconditional or conditional jump/goto statements are considered as leaders.

For each leader thus determined its basic block contains itself and all instructions up to excluding the next leader.

Partitioning Intermediate Code into Basic Blocks:

Any given code can be partitioned into basic blocks using the following rules-

Rule-01:

Determining Leaders-

Following statements of the code are called as Leaders—

- First statement of the code.
- Statement that is a target of the conditional or unconditional goto statement.
- Statement that appears immediately after a goto statement.

Rule-02:

Determining Basic Blocks-

- All the statements that follow the leader (including the leader) till the next leader appears form one basic block.
- The first statement of the code is called as the first leader.
- The block containing the first leader is called as Initial block.

Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.

The nodes of the flow graph are basic blocks. It has a distinguished initial node.

Flow Graph:

Once an intermediate code program is divided into basic blocks, via a flow graph, we represent the flow of control between them. The nodes of the flow graph are the basic blocks.

Rules for Flow Graph:

- The basic block are the nodes to the flow graph.
- The block whose leader is the first statement is called initial block.
- There is a directed edge from block B1 and block B2 if B2 immediately follows B1 in the given sequence, we can say that B1 is a predecessor of B2.

Example:

Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

1. `prod := 0`
2. `i := 1`
3. `t1 := 4*i`
4. `t2 := a[t1]`
5. `t3 := 4*i`
6. `t4 := b[t3]`

7. $t5 := t2 * t4$
8. $t6 := \text{prod} + t5$
9. $\text{prod} := t6$
10. $t7 := I + 1$
11. $i := t7$
12. $\text{if}(i \leq 20) \text{ goto}(3)$

Three Address Code & Leaders:

1. $\text{prod} := 0$ (Leader)
2. $i := 1$
3. $t1 := 4 * i$ (Leader)
4. $t2 := a[t1]$
5. $t3 := 4 * i$
6. $t4 := b[t3]$
7. $t5 := t2 * t4$
8. $t6 := \text{prod} + t5$
9. $\text{prod} := t6$
10. $t7 := i + 1$
11. $i := t7$
12. $\text{if}(i \leq 20) \text{ goto}(3)$

Here, the leaders are 1 and 3.

Basic block:

1) $\text{prod} := 0$ 2) $i := 1$

(3) $t1 := 4 * i$ (4) $t2 := a[t1]$ (5) $t3 := 4 * i$ (6) $t4 := b[t3]$ (7) $t5 := t2 * t4$ (8) $t6 := \text{prod} + t5$ (9) $\text{prod} := t6$ (10) $t7 := i + 1$ (11) $i := t7$ (12) $\text{if}(i \leq 20) \text{ goto}(3)$

Fig: Basic Block

Flow graph:

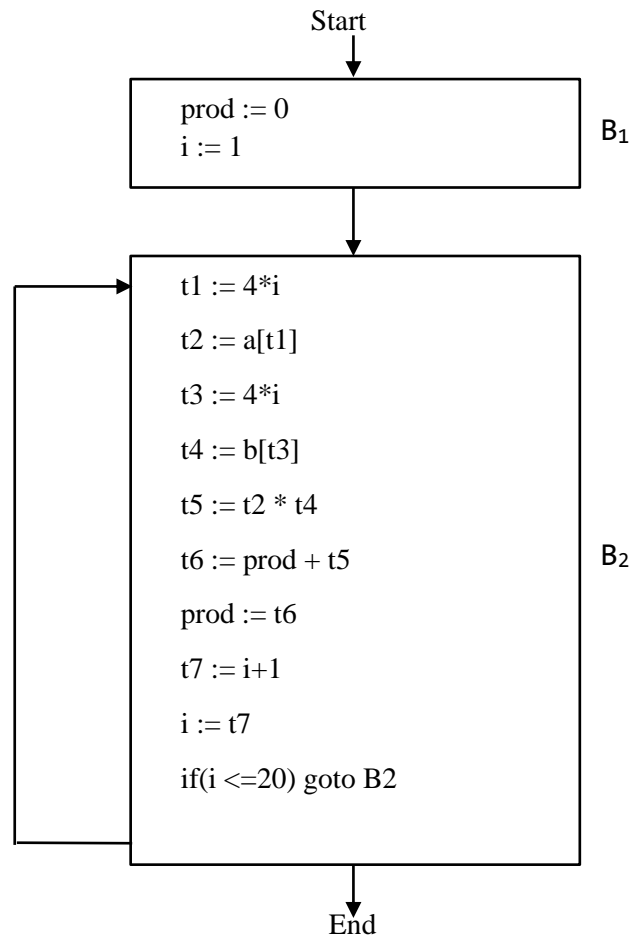


Fig: Flow Graph

Example:

Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

```
for i=1 ... 10 do
    for j=1 ... 10 do
        a[i,j] = 0.0;
    for i=1 ... 10 do
        a[i,i] = 1.0;
```

Three Address Code & Leaders:

```
1) i = 1                (Leader)
2) j = 1                (Leader)
3) t1 = 10 * i          (Leader)
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1           (Leader)
11) if i <= 10 goto (2)
12) i = 1               (Leader)
13) t5 = i - 1          (Leader)
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Here, the leaders are 1, 2, 3, 10, 12 and 13.

Basic Block:

(1) i=1

(2) j=1

(3) t₁=10*i
(4) t₂=t₁+j
(5) t₃=8*t₂
(6) t₄=t₃-88
(7) a[t₄]=0.0
(8) j=j+1
(9) if j<=10 goto (3)

(10) i=i+1
(11) if i<=10 goto (2)

(12) i=1

(13) t₅=i-1
(14) t₆=88* t₅
(15) a[t₆]=1.0
(16) i=i+1
(17) if i<=10 goto(13)

Fig: Basic Block

Flow Graph:

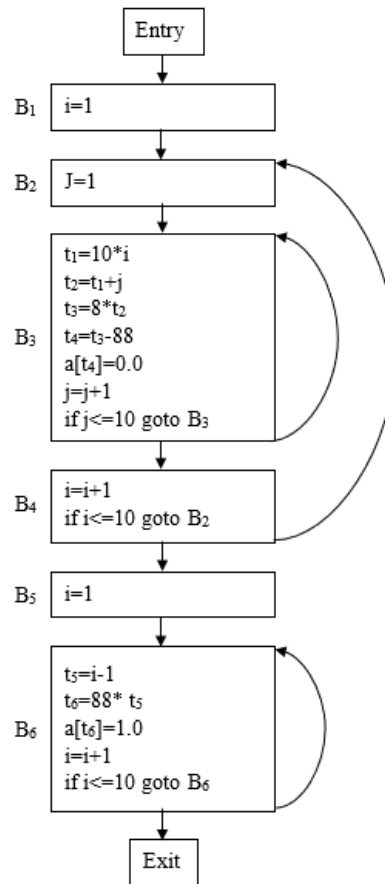


Fig: Flow Graph

Exercise:

Question1: Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

```
begin
  prod := 0;
  i := 0;
  do begin
    prod := prod + a[i] * b[i];
    i := i + 1;
  end
  while I <= 20
end
```

Question2: Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

```
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        c[i] [j] = 0.0;  
for (i=0; i<n; i++)  
    for (j=0; j<n; j++)  
        for (k=0; k<n; k++)  
            c[i] [j] = c[i] [j] + a[i] [k]*b[k] [j] ;
```

Question3: Construct Three Address Code, find & mark leaders, construct basic block and flow graph from the following given code.

```
for (i=2; i<=n; i++)  
    a[i] = TRUE;  
count = 0;  
s = sqrt (n) ;  
for (i=2; i<=s; i++)  
    if (a[i]) /* i has been found to be a prime */ {  
        count++ ;  
        for (j=2*i; j<=n; j = j+i)  
            a[j] = FALSE; /* no multiple of i is a prime */  
    }  
}
```

Questions:

1. What is Basic Block?
2. Write down the rules for determining leaders, basic blocks and flow graphs.

Chapter - 11

Code Optimization

Code optimization is any method of code modification to improve code quality and efficiency. A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.

Why do we need Code Optimization?

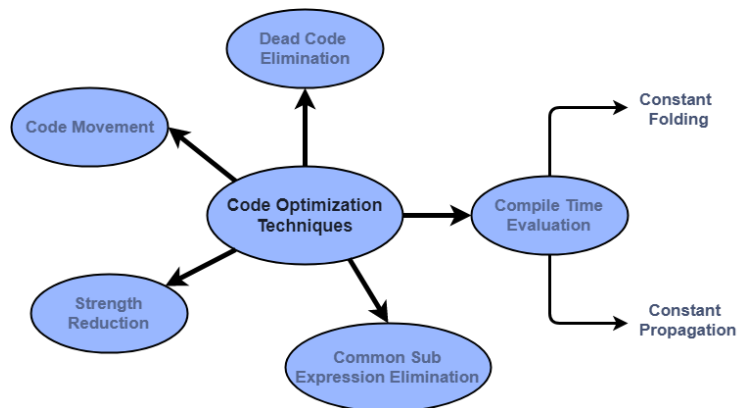
An important phase of compiler related to improving the quality of codes by removing unnecessary lines as well as rearranging the statements of the code

This makes code run faster and hence optimized code gives better performance. Code optimization allows consumption of fewer resources. (i.e., CPU, Memory), which results in faster running machine code. Optimized code also uses memory efficiently.

Code Optimization Techniques:

Important code optimization techniques are-

1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction



1. Compile Time Evaluation:

Two techniques that falls under compile time evaluation are-

A) Constant Folding-

In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Example:

Circumference of Circle = $(22/7) \times \text{Diameter}$

Here,

- This technique evaluates the expression $22/7$ at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

B) Constant Propagation

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

Example

$\text{pi} = 3.14$

$\text{radius} = 10$

$\text{Area of circle} = \text{pi} \times \text{radius} \times \text{radius}$

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.

- It then evaluates the expression $3.14 \times 10 \times 10$.
- The expression is then replaced with its result 314.
- This saves the time at run time.

2. Common Sub-Expression Elimination:

The expression that has been already computed before and appears again in the code for computation is called as **Common Sub-Expression**.

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Example:

Code Before Optimization	Code After Optimization
S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // Redundant Expression S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

3. Code Movement:

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example

Code Before Optimization	Code After Optimization
<pre>for (int j = 0; j < n; j ++) { x = y + z; a[j] = 6 x j; }</pre>	<pre>x = y + z; for (int j = 0; j < n; j ++) { a[j] = 6 x j; }</pre>

4. Dead Code Elimination:

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example:

Code Before Optimization	Code After Optimization
<pre>i = 0 ; if (i == 1) { a = x + 5 ; }</pre>	<pre>i = 0 ;</pre>

5. Strength Reduction:

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.
-

Example:

Code Before Optimization	Code After Optimization
<pre>B = A x 2</pre>	<pre>B = A + A</pre>

Here,

- The expression “ $A \times 2$ ” is replaced with the expression “ $A + A$ ”.
- This is because the cost of multiplication operator is higher than that of addition operator.

Questions:

- Why we use Code Optimization?
- What is the benefit of Code Optimization?
- Describe all the techniques of Code Optimization with example.

Fall-2020

Booklet

-By A big Zero

Learn coding in a new way.

A LAB Handbook on Compiler Design

BY

Md. Mosfikur Rahman
ID: 181-15-2065

Akash Ahmed
ID: 181-15-1714

Md.kawser Ahamed
ID: 181-15-1722

Md. Abtab Uddin Akib
ID:181-15-2000

Tania sultana khanom
ID: 181-15-1945

Mahfuja Ferdousi Mahin
ID: 181-15-1860

Rakibul Hassan Raja
ID: 181-15-2041

Md. Hasan Imam
ID: 181-15-2027

This handbook presents compiler's working procedure in respectively manual and automatic optimization with C programming.

Supervised By
Mushfiqur Rahman
Lecturer
Department of CSE
Daffodil International University



DAFFODIL INTERNATIONAL UNIVERSITY

Dhaka, Bangladesh

DECLARATION

We hereby declare that, this handbook has been made by us under the supervision of Mr. Mushfiqur Rahman, Lecturer, Department of CSE, Daffodil International University. We also declare that neither this handbook nor any part of this handbook has been submitted or published elsewhere.

Supervised by:

Mushfiqur Rahman

Lecturer

Department of CSE

Daffodil International University

Submitted by:

Md. Mosfikur Rahman

ID: 181-15-2065

Department of CSE

Daffodil International University

Akash Ahmed

ID: 181-15-1714

Department of CSE

Daffodil International University

Md. kawser Ahamed

ID: 181-15-1722

Department of CSE

Daffodil International University

Md. Abtab Uddin Akib

ID: 181-15-2000

Department of CSE

Daffodil International University

Tania sultana khanom

ID: 181-15-1945

Department of CSE

Daffodil International University

Mahfuja Ferdousi Mahin

ID: ID: 181-15-1860

Department of CSE

Daffodil International University

Rakibul Hassan Raja

ID: 181-15-2041

Department of CSE

Daffodil International University

Md. Hasan Imam

ID: 181-15-2027

Department of CSE

Daffodil International University

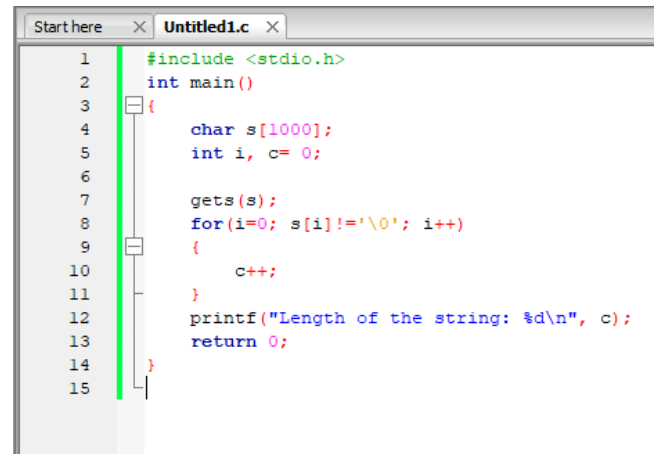
Table of Contents

No.	Page No.
1. Write a program that will count the length of a string_____	1
2. Write a C program that will count the number of white spaces from a string_____	1
3. C program that will remove white space from a string_____	2
4. Write a program that will count vowel, consonant, and digit_____	2
5. Write two C program that will tokenize a string. (using strtok() and also without using any library function)_____	4
6. Write a C program that will take multiple lines as input and count the number of Lines_____	6
7. Write a C program that will take multiple lines as input and identify the comments if there any_____	7
8. Write a C program that will take multiple lines as input and remove the single line/multiple line comments if there any_____	8
9. Write a C program that will identify the articles from a given input string and count the articles_____	11
10. To Write a C program to test whether a given identifier is valid or not_____	13
11. Write a C program that will Count and show the max frequency of a word in a string_____	15
12. Project: Infix to postfix converter_____	16

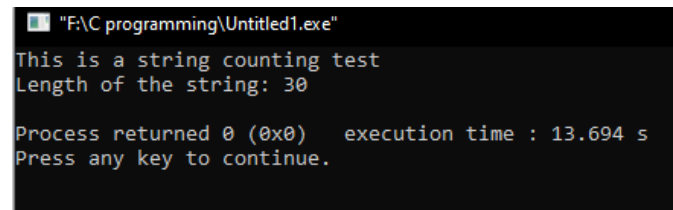
Problem 01: Write a program that will **count the length** of a string.

```
#include <stdio.h>
int main()
{
    char s[1000];
    int i, c= 0;

    gets(s);
    for(i=0; s[i]!='\0'; i++)
    {
        c++;
    }
    printf("Length of the string: %d\n", c);
    return 0;
}
```

A screenshot of a code editor window titled 'Untitled1.c'. The code is a C program to count the length of a string. It includes the standard input/output header, defines a main function, declares a character array 's' of size 1000 and an integer 'c' initialized to 0. It uses 'gets(s)' to read a string, then a 'for' loop to iterate through each character until the null terminator '\0' is reached, incrementing 'c' for each character. Finally, it prints the length using 'printf' and returns 0.

```
1  #include <stdio.h>
2  int main()
3  {
4      char s[1000];
5      int i, c= 0;
6
7      gets(s);
8      for(i=0; s[i]!='\0'; i++)
9      {
10         c++;
11     }
12     printf("Length of the string: %d\n", c);
13     return 0;
14 }
15
```

A screenshot of a Windows command prompt window titled '"F:\C programming\Untitled1.exe"'. It shows the output of the program: 'This is a string counting test' followed by 'Length of the string: 30'. Below this, it displays 'Process returned 0 (0x0) execution time : 13.694 s' and 'Press any key to continue.'.

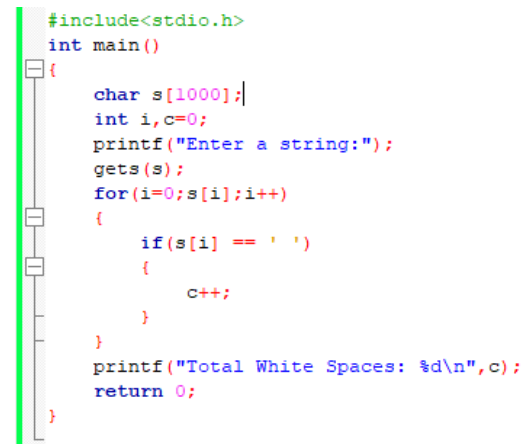
```
"F:\C programming\Untitled1.exe"
This is a string counting test
Length of the string: 30

Process returned 0 (0x0)   execution time : 13.694 s
Press any key to continue.
```

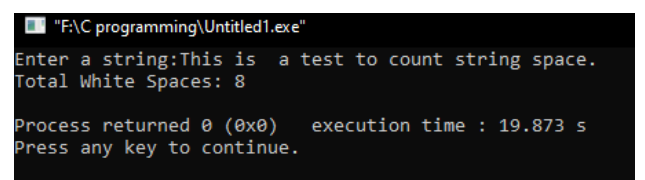
Description: To count the length of a string we used gets to scan the string then we counter the characters using a loop until the end of the string ('\0') and printed the value of the count.

Problem 02: Write a C program that will **count the number of white spaces** from a string.

```
#include<stdio.h>
int main()
{
    char s[1000];
    int i,c=0;
    printf("Enter a string:");
    gets(s);
    for(i=0;s[i];i++)
    {
        if(s[i] == ' ')
        {
            c++;
        }
    }
    printf("Total White Spaces: %d\n",c);
    return 0;
}
```

A screenshot of a code editor window showing the implementation of Problem 02. The code includes the standard input/output header, defines a main function, declares a character array 's' of size 1000 and an integer 'c' initialized to 0. It prompts the user to 'Enter a string:' and reads the input with 'gets(s)'. A 'for' loop iterates through each character of the string. Inside the loop, an 'if' statement checks if the current character is a space (' '). If it is, 'c' is incremented. After the loop, the total number of white spaces is printed using 'printf' and the function returns 0.

```
#include<stdio.h>
int main()
{
    char s[1000];
    int i,c=0;
    printf("Enter a string:");
    gets(s);
    for(i=0;s[i];i++)
    {
        if(s[i] == ' ')
        {
            c++;
        }
    }
    printf("Total White Spaces: %d\n",c);
    return 0;
}
```

A screenshot of a Windows command prompt window titled '"F:\C programming\Untitled1.exe"'. It shows the user inputting 'This is a test to count string space.' in response to the prompt 'Enter a string:'. The output is 'Total White Spaces: 8'. Below this, it displays 'Process returned 0 (0x0) execution time : 19.873 s' and 'Press any key to continue.'.

```
"F:\C programming\Untitled1.exe"
Enter a string:This is a test to count string space.
Total White Spaces: 8

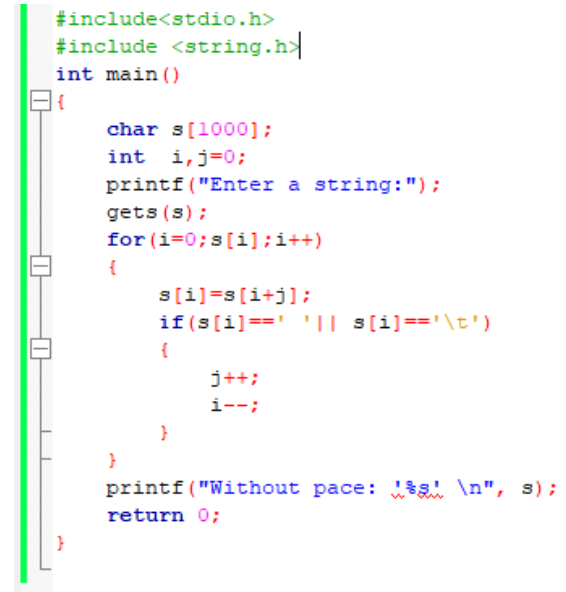
Process returned 0 (0x0)   execution time : 19.873 s
Press any key to continue.
```

```
}
```

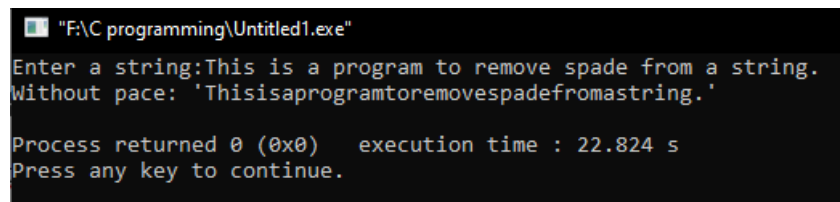
Description: To count the spaces in a string we used a for loop and counter the characters if they were a space (' '). And printed the count value.

Problem 03: C program that will **remove white space** from a string.

```
#include<stdio.h>
#include <string.h>
int main()
{
    char s[1000];
    int i,j=0;
    printf("Enter a string:");
    gets(s);
    for(i=0;s[i];i++)
    {
        s[i]=s[i+j];
        if(s[i]==' '|| s[i]=='\t')
        {
            j++;
            i--;
        }
    }
    printf("Without space: '%s' \n", s);
    return 0;
}
```



```
#include<stdio.h>
#include <string.h>
int main()
{
    char s[1000];
    int i,j=0;
    printf("Enter a string:");
    gets(s);
    for(i=0;s[i];i++)
    {
        s[i]=s[i+j];
        if(s[i]==' '|| s[i]=='\t')
        {
            j++;
            i--;
        }
    }
    printf("Without space: '%s' \n", s);
    return 0;
}
```



```
"F:\C programming\Untitled1.exe"
Enter a string:This is a program to remove spade from a string.
Without space: 'Thisisaprogramtoremovespadefromastring.'

Process returned 0 (0x0)   execution time : 22.824 s
Press any key to continue.
```

Description: Here 1st we took a string then used a for loop. Then we used a value j = 0 and added in it with the string [i]. After that we checked if there is any space or whitespace in the string. If there was any then increase the value of j and decrease the value of i by 1 to skip the space and create a new string as s. By overwriting the value of i in the string we successfully deleted all the spaces in a string.

Problem 04: Write a program that will count vowel, consonant, and digit.

```
#include <stdio.h>
#include <string.h>
int main()
```

```

{
    char s[100];
    int i,vowels=0,consonants=0,digites=0,character=0;

    printf("Enter the string : ");
    gets(s);

    for(i=0;s[i];i++)
    {
        if(s[i]>=48 && s[i]<=57)
            digites++;

        else if((s[i]>=65 && s[i]<=90) || (s[i]>=97 && s[i]<=122))
        {
            if(s[i]=='a' ||
s[i]=='e' || s[i]=='i' || s[i]=='o' || s[i]=='u' || s[i]=='A' || s[i]=='E' || s[i]=='I' ||
s[i]=='O' || s[i]=='U')
                vowels++;

            else
                consonants++;
        }
        else character++;
    }

    printf("vowels = %d\n",vowels);
    printf("consonants = %d\n",consonants);
    printf("Digits = %d\n",digites);
    printf("characters = %d\n",character);

    return 0;
}

```

The screenshot shows a C++ IDE with the following code in the editor:

```

1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char s[100];
6     int i,vowels=0,consonants=0,digites=0,character=0;
7
8     printf("Enter the string : ");
9     gets(s);
10
11     for(i=0;s[i];i++)
12     {
13         if(s[i]>=48 && s[i]<=57)
14             digites++;
15
16         else if((s[i]>=65 && s[i]<=90) || (s[i]>=97 && s[i]<=122))
17         {
18             if(s[i]=='a' || s[i]=='e' || s[i]=='i' || s[i]=='o' || s[i]=='u' || s[i]=='A' || s[i]=='E' || s[i]=='I' || s[i]=='O' || s[i]=='U')
19                 vowels++;
20             else
21                 consonants++;
22         }
23         else character++;
24     }
25
26     printf("vowels = %d\n",vowels);
27     printf("consonants = %d\n",consonants);
28     printf("Digits = %d\n",digites);
29     printf("characters = %d\n",character);
30
31     return 0;
32 }
33

```

The output window shows the following results:

```

"D:\C programming\Untitled1.exe"
Enter the string : This is 1 sentence to check !@#$%^&*
vowels = 7
consonants = 14
Digits = 1
characters = 14

Process returned 0 (0x0)   execution time : 37.929 s
Press any key to continue.

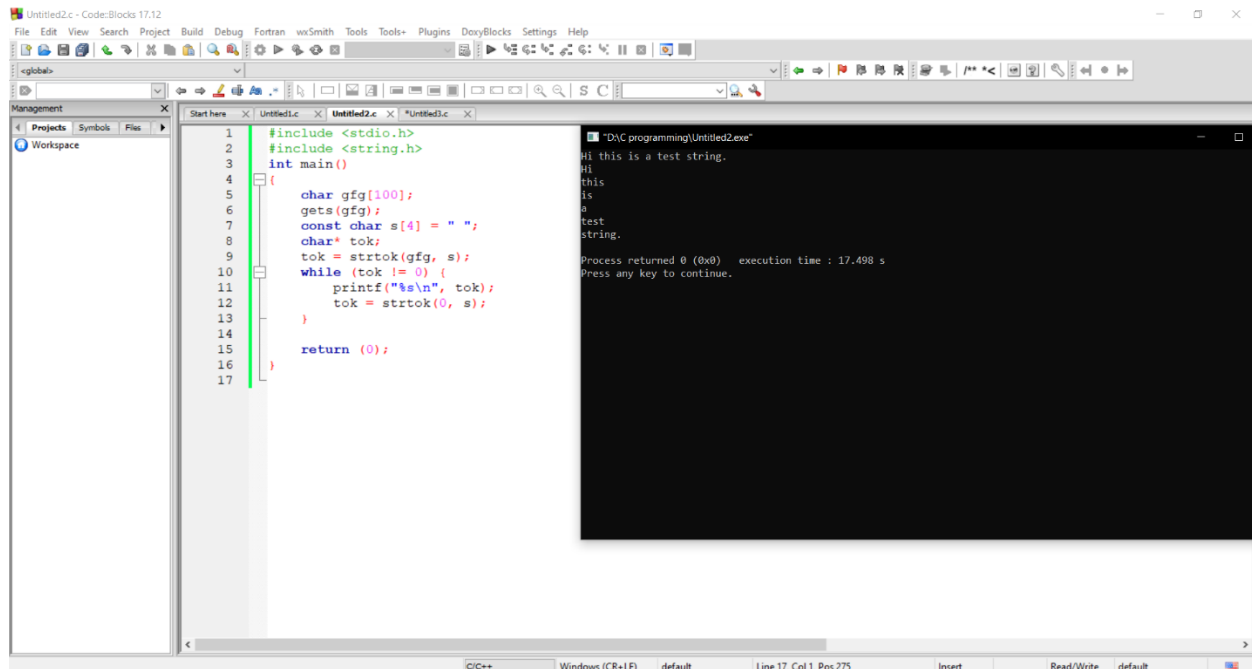
```


Description: Using ascii value we can simplify our code as we don't need to think of all the characters. We can use range in for loop to easily detect all the characters (vowels, consonant, digits and special characters.)

Problem 05: Write two C program that will tokenize a string. (**using strtok()** and also **without using any library function**)

Using strtok() function:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char gfg[100];
    gets(gfg);
    const char s[4] = " ";
    char* tok;
    tok = strtok(gfg, s);
    while (tok != 0) {
        printf("%s\n", tok);
        tok = strtok(0, s);
    }
    return (0);
}
```



```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char gfg[100];
6     gets(gfg);
7     const char s[4] = " ";
8     char* tok;
9     tok = strtok(gfg, s);
10    while (tok != 0) {
11        printf("%s\n", tok);
12        tok = strtok(0, s);
13    }
14    return (0);
15 }
16
17
```

Process returned 0 (0x0) execution time : 17.498 s
Press any key to continue.

Description: strtok() is a function to cut a string in to tokens. We used this function to shortly split a string into tokens.

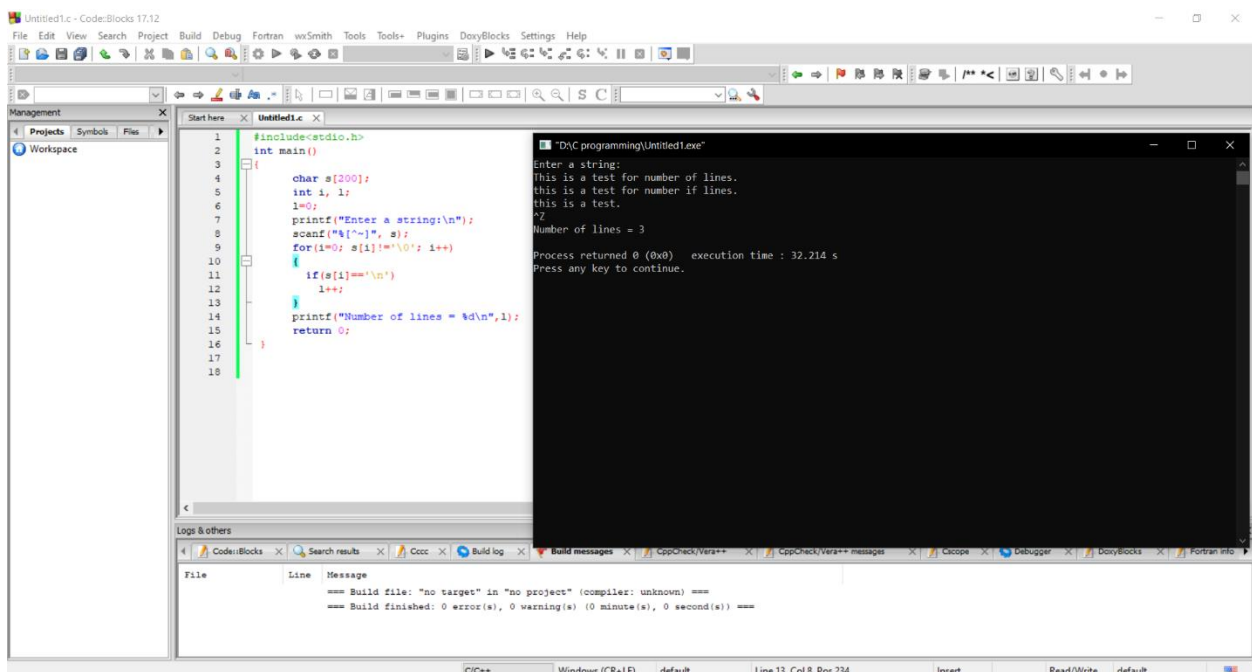
Without using any library function:

```
#include<stdio.h>
int main()
{
    int i=0;
    char s[50];
    printf("Enter String : ");
    gets(s);
    while(s[i]!='\0')
    {
        if(s[i]==' ')
        {
            s[i]='\n';
        }
        i++;
    }
    printf("%s",s);
    return 0;
}
```

Description: to split a string without using any function we only replace the space with line break (back slash). It is a convenient process to convert a string into small tokens.

Problem 06: Write a C program that will take multiple lines as input and count the number of Lines.

```
#include<stdio.h>
int main()
{
    char s[200];
    int i, l;
    l=0;
    printf("Enter a string:\n");
    scanf("%[^\n]", s);
    for(i=0; s[i]!='\0'; i++)
    {
        if(s[i]=='\n')
            l++;
    }
    printf("Number of lines = %d\n",l);
    return 0;
}
```



The screenshot shows the Code::Blocks IDE with a C program open. The program prompts the user to "Enter a string:" and then reads input using `scanf("%[^\n]", s);`. The input entered is "This is a test for number of lines. this is a test for number if lines. this is a test." followed by a carriage return character (ASCII 10). The program then prints "Number of lines = 3". The output window shows the execution details, including the return code (0) and execution time (32.214 s).

Discussion: To solve this problem we need to eliminate the line breaks in enter that stopes the whole program. So, we used `"%[^\n]"` in `scanf()` so that it will take all kind of strike from the keyboard as input except `^~`. When we put `^~` as input the whole program stopes taking input and start executing all. Then we just have to out how many new lines are in the whole sentence. Then we will be able to count the number of lines in the text that we have entered as input.

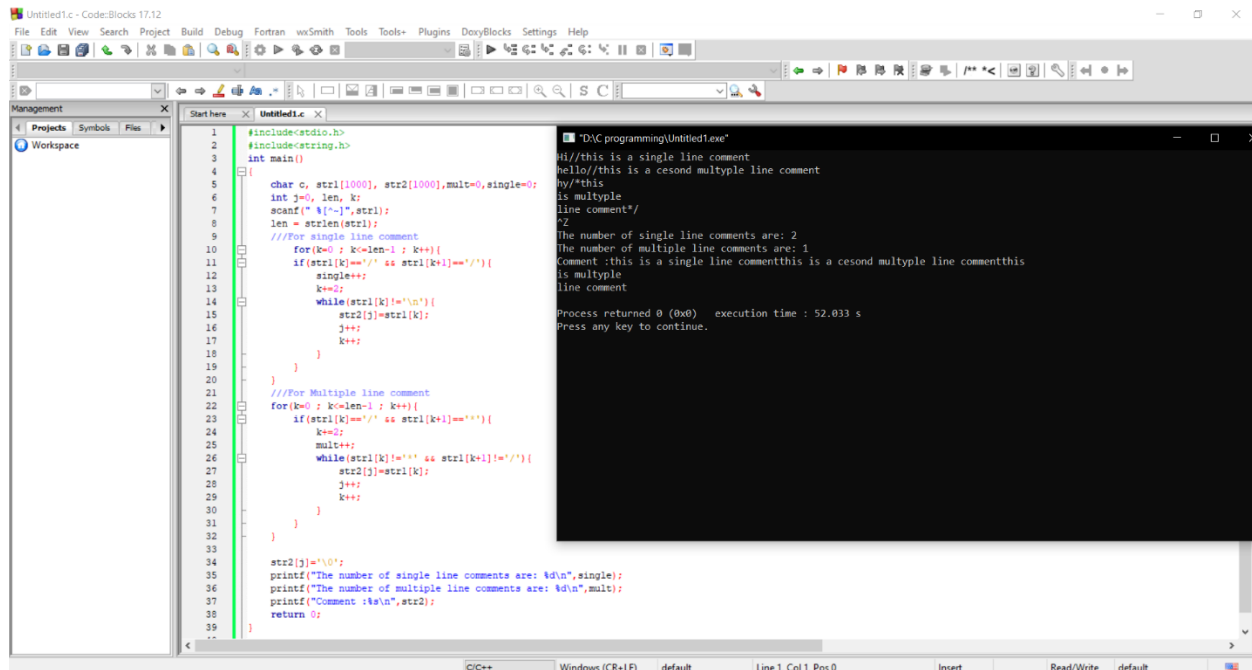
Problem 07: Write a C program that will take **multiple lines as input** and **identify the comments** if there any.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char c, str1[1000], str2[1000],mult=0,single=0;
    int j=0, len, k;
    scanf("%[^\n]",str1);
    len = strlen(str1);
    ///For single line comment
    for(k=0 ; k<=len-1 ; k++){
        if(str1[k]=='/' && str1[k+1]=='/'){
            single++;
            k+=2;
            while(str1[k]!='\n'){
                str2[j]=str1[k];
                j++;
                k++;
            }
        }
    }
    ///For Multiple line comment
    for(k=0 ; k<=len-1 ; k++){
        if(str1[k]=='/' && str1[k+1]=='*'){
            k+=2;
            mult++;
            while(str1[k]!='*' && str1[k+1]!='/'){
                str2[j]=str1[k];
                j++;
                k++;
            }
        }
    }

    str2[j]='\0';
    printf("The number of single line comments are: %d\n",single);
    printf("The number of multiple line comments are: %d\n",mult);
    printf("Comment :%s\n",str2);
    return 0;
}
```

Discussion: Here we took "%[^\n]" for entering multiple line as input. Then we used conditions to find the comment and stored them in another comment. Then we printed the whole string to show to comments in the input text.

As the question was to detect single and multiple line comment so we counted the single and multiple line comment and printed the number also.



Problem 08: Write a C program that will take multiple lines as input and remove the single line/multiple line comments if there any.

```

#include <stdio.h>
#include <stdlib.h>

FILE *fp , *fp2;

void check_comment(char c)
{
    char ch;
    if( c == '/')
    {
        if((ch=fgetc(fp))=='*')
            block_comment();
        else if( ch == '/')
        {
            single_comment();
        }
        else
        {
            fputc(c, fp2);
            fputc(ch, fp2);
        }
    }
    else

```

With_Comment.txt - Notepad

File Edit Format View Help

Hi//this is a comment

Hello/*This is a test
multiline
comment for*/

TEST.//test comment

Ok../* This is a test
multiline
comment for*/

Without_Comment.txt - Notepad

File Edit Format View Help

Hi
Hello

TEST.
Ok..

```

        fputc(c, fp2);
    }
void block_comment()
{
    char ch1, ch2;
    while((ch1=fgetc(fp))!=EOF)
    {
        if(ch1=='*')
        {
            ch2=fgetc(fp);

            if(ch2=='/')
                return;
        }
    }
}

void single_comment()
{
    char ch1;
    while((ch1=fgetc(fp))!=EOF)
    {
        if(ch1=='\n')
            return;
    }
}

int main(void)
{
    char c;
    fp = fopen ("With_Comment.txt", "r") ;
    fp2 = fopen ("Wothout_Comment.txt", "w") ;
    while((c=fgetc(fp))!=EOF)
        check_comment(c);
    fclose(fp);
    fclose(fp2);
    return 0;
}

```

Discussion: Using files we can first catch the characters by exploring them and testing them to be a comment or not then putting those characters those aren't comment in another file. Using fgetc() function we selected the characters one by one and putting them in "mynewfile.txt" file. We can solve it without using the file too. It is given below.

Second way:

```

#include<stdio.h>
void rcomment(int c);

```

```

void incomment(void);
void incomment2(void);
void echo_quote(int c);
int main(void)
{
    int c,d;
    printf(" To Remove Comment:\n");
    while((c=getchar())!=EOF)
        rcomment(c);
    return 0;
}
void rcomment(int c)
{
    int d;

    if( c == '/')
    {
        if((d=getchar())=='*')
            incomment();
        else if( d == '/')
        {
            incomment2();
        }
        else
        {
            putchar(c);
            putchar(d);
        }
    }
    else if( c == '\\' || c == '"')
        echo_quote(c);
    else
        putchar(c);
}

void incomment()
{
    int c,d;

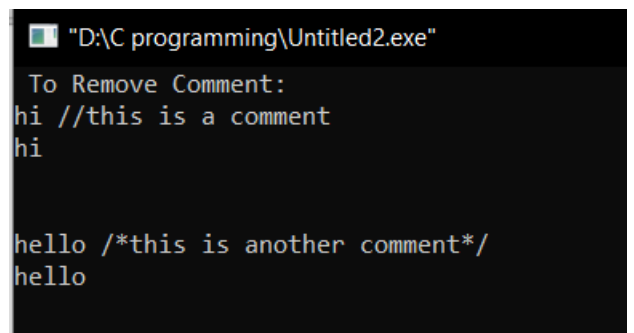
    c = getchar();
    d = getchar();

    while(c!='*' || d != '/')
    {
        c = d;
        d = getchar();
    }
}

void incomment2()

```

Output:



```

D:\C programming\Untitled2.exe
To Remove Comment:
hi //this is a comment
hi

hello /*this is another comment*/
hello

```

```

{
    int c,d;

    c = getchar();
    d = getchar();

    while(c!='\n' )
    {
        c =d;
        d = getchar();
    }
}

void echo_quote(int c)
{
    int d;

    putchar(c);

    while((d=getchar())!=c)
    {
        putchar(d);

        if(d == '\\')
            putchar(getchar());
    }
    putchar(d);
}

```

Problem 09: Write a C program that will identify the articles from a given input string and count the articles.

```

#include<stdio.h>
#include<string.h>
int main()
{
    int i, a=0, an=0, the=0;
    char ch[100];
    gets(ch);
    for(i=0; ch[i]!='\0'; i++){
        if((ch[i]<65 || ch[i]>90) && (ch[i]<97 || ch[i]>122)){
            if(ch[i-1]=='e' && ch[i-2]=='h' && (ch[i-3]=='T' || ch[i-3]=='t'))&&(ch[i-4]<65 || ch[i-4]>90||i-3==0) && (ch[i-4]<97 || ch[i-4]>122||i-3==0))
                the++;
            else if(ch[i-1]=='n' && (ch[i-2]=='a' || ch[i-2]=='A')&&(ch[i-3]<65 || ch[i-3]>90||i-2==0) && (ch[i-3]<97 || ch[i-3]>122||i-2==0))
                an++;
            else if((ch[i-1]=='a' || ch[i-1]=='A')&&(ch[i-2]<65 || ch[i-2]>90||i-1==0) && (ch[i-2]<97 || ch[i-2]>122||i-1==0))

```



```

        a++;
    }
}
printf("A - %d\nAn - %d\nThe - %d\n",a, an, the);
return 0;
}

```

Discussion: To solve this problem first we thought of collecting the ascii values of all character so that we can try to make a range of all alphabetical characters. Then we created a range of it. After that for “the” and “The” we tried to find a logic that if the first character was “T” or “t” second character “h” and the third character “e” if this logic is granter then we compared it with the ascii values to find the space after all these characters. If the logics is correct then we counter the values of “the”. The same way we also counter the values of “a” and “an” for the program.

```

1  #include<stdio.h>
2  #include<string.h>
3  int main()
4  {
5      int i, a=0, an=0, the=0;
6      char ch[100];
7      gets(ch);
8      for(i=0; ch[i]!='\0'; i++){
9          if((ch[i]<65 || ch[i]>90) && (ch[i]<97 || ch[i]>122)){
10             if(ch[i-1]=='e' && ch[i-2]=='h' && (ch[i-3]=='T' || ch[i-3]=='t')){
11                 the++;
12             }
13             else if(ch[i-1]=='n' && (ch[i-2]=='a' || ch[i-2]=='A')){
14                 an++;
15             }
16             else if((ch[i-1]=='a' || ch[i-1]=='A') && (ch[i-2]<65 || ch[i-2]>90 || i-1==0) && (ch[i-2]<97 || ch[i-2]>122 || i-1==0)){
17                 a++;
18             }
19         }
20     }
21     printf("A - %d\nAn - %d\nThe - %d\n",a, an, the);
22     return 0;
23 }

```

Terminal Output:

```

D:\C programming\Untitled2.exe
An ant bit the boy and a bird saw the event.
A - 1
An - 1
The - 2
Process returned 0 (0x0)   execution time : 1.785 s
Press any key to continue.

```

Problem 10: To Write a C program to test whether a given identifier is valid or not.

```

#include <stdio.h>
#include <mem.h>
#include <ctype.h>
int main() {
    int i, flag = 0;

```

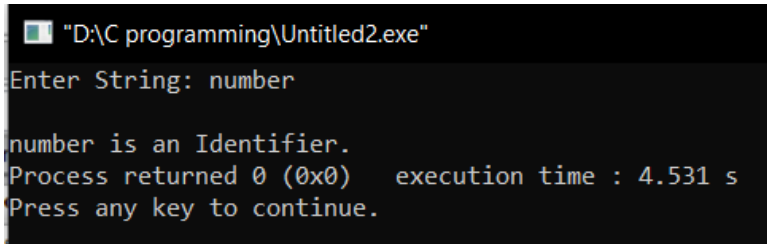
```

char keyword[32][100] = { "double", "else", "enum", "extern", "float",
"switch", "typedef", "union",
"const", "continue", "default", "if", "int", "long", "register", "return",
"for", "goto", "short", "signed", "auto", "break", "case", "char",
"sizeof", "static", "struct", "do", "unsigned", "void",
"volatile", "while"}, a[100];
printf("Enter String: ");
gets(a);
for(i = 0; i < 32; ++i){
    if((strcmp(keyword[i], a) == 0)){
        flag = 1;
    }
}
if(flag == 1){
    printf("\n%s is keyword.", a);
}
else {
    flag = 0;
    if((a[0] == '_' || (isalpha(a[0]) != 0))){
        for(i = 1; a[i] != '\0'; ++i){
            if((isalnum(a[i]) == 0) && (a[i] != '_')){
                flag = 1;
            }
        }
    }
    else{
        flag = 1;
    }
}
if(flag == 0){
    printf("\n%s is an Identifier.", a);
}
else{
    printf("\n%s is Not an Identifier.", a);
}

return 0;
}

```

Outputs:



```

D:\C programming\Untitled2.exe
Enter String: number

number is an Identifier.
Process returned 0 (0x0)   execution time : 4.531 s
Press any key to continue.

```

```
"D:\C programming\Untitled2.exe"
Enter String: 1number

1number is Not an Identifier.
Process returned 0 (0x0)   execution time : 2.569 s
Press any key to continue.

"D:\C programming\Untitled2.exe"
Enter String: int

int is keyword.
int is Not an Identifier.
Process returned 0 (0x0)   execution time : 1.628 s
Press any key to continue.
```

Discussion: To solve this problem we first need to know the what is an identifier. Identifier refers to name given to entities such as variables, functions, structures etc. Identifiers must be unique. There are some rules to make identifiers.

Rules for an Identifier

- An Identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore (_).
- The first character of an identifier can only contain alphabet (a-z, A-Z) or underscore (_).
- Identifiers are also case sensitive in C. For example, name and Name are two different identifiers in C.
- Keywords are not allowed to be used as Identifiers.
("int", "float", "break", "long", "char", "for", "if", "switch", "else", "while")
- No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

So, to solve this problem we need to cancel out all the rules from the characters from the input.

We took all the keywords in a 2D array then canceled them out then the number and the underscore.

Problem 01: Write a C program that will Count and show the max frequency of a word in a string.

```
#include<stdio.h>
#include<string.h>

int main() {
    char str[100][100];
```

```

int i, j = 1, x, b = 0, c = 0, d = 0, e = 0;

printf("Enter String:\n");
gets(str[0]);

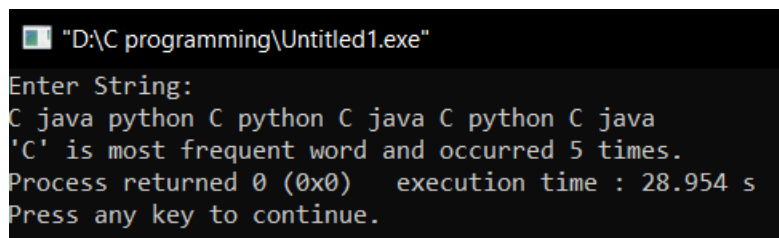
x = strlen(str[0]);

for (i = 0; i < x; ++i) {
    if (str[0][i] != ' ') {
        ++c;
        while(str[0][i] != ' ' && str[0][i] != '\0') {
            str[j][e++] = str[0][i++];
        }
        ++j;
        e = 0;
    }
}

for(i = 1; i <= c; ++i) {
    for(j = 1; j <= c; ++j) {
        if (strcmp(str[i], str[j]) == 0) {
            ++d;
        }
    }
    if (d > b) {
        x = i;
        b = d;
    }
    d = 0;
}
printf("' %s' is most frequent word and occurred %d times.", str[x], b);
return 0;
}

```

Outputs:



```

"D:\C programming\Untitled1.exe"
Enter String:
C java python C python C java C python C java
'C' is most frequent word and occurred 5 times.
Process returned 0 (0x0)   execution time : 28.954 s
Press any key to continue.

```

Discussion: To solve this problem first we need to take a string as input. Then we will have to segment out string into while space and end of the line in a loop to find the frequent word in the given string by taking string's length with `strlen()` function. After that in the second loop we will have count how many times that frequent word occurs.

Project: Infix to postfix converter

```
#include<stdio.h>
#include<ctype.h>

char stack[100];
int top = -1;

void push(char ch)
{
    stack[++top] = ch;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *pointer, x;
    printf("Enter a infix expression : ");
    scanf("%s",exp);
    printf("\n");

    printf("The postfix expression is: ");
    pointer = exp;

    while(*pointer != '\0')
    {
```

```

    if(isalnum(*pointer))
        printf("%c ",*pointer);
    else if(*pointer == '(')
        push(*pointer);
    else if(*pointer == ')')
    {
        while((x = pop()) != '(')
            printf("%c ", x);
    }
    else
    {
        while(priority(stack[top]) >= priority(*pointer))
            printf("%c ",pop());
        push(*pointer);
    }
    pointer++;
}

while(top != -1)
{
    printf("%c ",pop());
}return 0;
}

```