DAYANANDA SAGAR UNIVERSITY

Vision

To be a Centre of excellence in education, research & training, innovation & entrepreneurship and to produce citizens with exceptional leadership qualities to serve national and global needs.

Mission

To achieve our objectives in an environment that enhances creativity, innovation and scholarly pursuits while adhering to our vision.

Values

The values that drive DSU and support its vision:

The Pursuit of Excellence

□ A commitment to strive continuously to improve ourselves and our systems with the aim of becoming the best in our field.

Fairness

□ A commitment to objectivity and impartiality, to earn the trust and respect of society.

Leadership

□ A commitment to lead responsively and creatively in educational and research processes.

Integrity and Transparency

□ A commitment to be ethical, sincere and transparent in all activities and to treat all individuals with dignity and respect.

Dayananda Sagar University

Laboratory Certificate

This is to certify that Mr./Ms.____bearing University Seat umber (USN)) _____________________________ has satisfactorily completed the ***COMPILER DESIGN AND SYSTEMS SOFTWARE LABORATORY(16CS373)*** prescribed by the University for the______semester, B. Tech. _______________branch of this university during the academic year

20__________20__________

Date: __________________

____

Signature of the Faculty in charge

Marks	
Maximum	**Obtained**

____

Signature of the Chairman

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

STUDENT PERFORMANCE REPORT

Sl. No	Date	Particulars	Marks Obtained	Initials of Staff
1a				
1b				
2a				
2b				
3a				
3b				
4				
5				
6				
7				
8				
9				
10				
11				

Internals	Lab Manual and Mini Project	Total Marks
		40

Signature of the Student **Signature of the Staff**

Course Objectives

The student should be made to:

- **Experiment** on the basic techniques of compiler construction and tools that can used to perform syntax-directed translation of a high-level programming language into an executable code.

- **Know** the implementation of assemblers, loaders and various parsing techniques.

- **Learn** how to optimize and effectively generate machine codes.

Course Outcomes

At the end of the course, the student should be able to:

1. **Understand** the working of lex and yacc compiler for debugging of programs.
2. **Understand** and define the role of lexical analyzer, absolute loader and symbol table.
3. **Learn** & use the new tools and technologies used for designing a compiler.
4. **Develop** program for solving parser problems.

INDEX

Lab Programs	**Page no**
1a. Program to count the number of characters, words, spaces and lines in a given input file.	
1b. Program to recognize and count the number of identifiers in a file.	
2a. Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.	
2b. Program to recognize whether a given sentence is simple or compound.	
3a. Program to count no of: i.+ve and –ve integers ii. +ve and –ve fractions	
3b. Program to count the no of „scanf‟ and „printf‟ statements in a C program. Replace them with „readf‟ and „writef‟ statements respectively.	
4.Program to evaluate arithmetic expression involving operators +,-,*,/	
5. Program to recognize a valid variable which starts with a letter, followed by any number of letters or digits.	
6. Program to recognize the strings using the grammar (a^nb^n ;n>=0)	
7. C Program to implement Pass1 of Assembler	
8. C Program to implement Absolute Loader	
9. C program to find the FIRST in context free grammar.	
10.C Program to implement Shift Reduce Parser for the given grammar E →E+E E→E*E E→(E) E → id	
11. C Program to implement code optimization techniques.	

1. INTRODUCTION TO LEX

Lex and YACC helps you write programs that transforms structured input. Lex generates C code for lexical analyzer whereas YACC generates Code for Syntax analyzer. Lexical analyzer is build using a tool called LEX. Input is given to LEX and lexical analyzer is generated.

Lex is a UNIX utility. It is a program generator designed for lexical processing of character input streams. Lex generates C code for lexical analyzer. It uses the **patterns** that match **strings in the input** and converts **the strings** to tokens. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer. The token descriptions that Lex uses are known as regular expressions.

1.1 Steps in writing LEX Program

1st Step: Using gedit create a file with extension l. For example: prg1.l

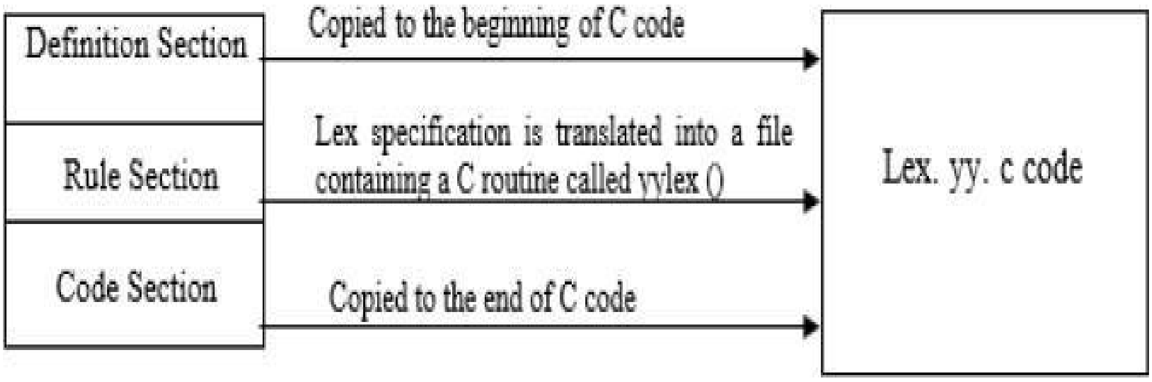
2nd Step: lex prg1.l

3rd Step: cc lex.yy.c –ll

4th Step: ./a.out

1.2 Structure of LEX source program

```
{definitions}
%%
{rules}
%%
{user subroutines/code section }
```


Definition Section
Rule Section
Code Section
Copied to the beginning of C code
Lex specification is translated into a file containing a C routine called yylex ()
Copied to the end of C code
Lex. yy. c code

%   is a delimiter to the mark the beginning of the Rule section. The second %% is optional, but the first is required to mark the beginning of the rules. The definitions and the code subroutines are often omitted.

LEX VARIABLES	
yyin	of the type FILE*. This point to the current file being parsed by the lexer.
yyout	of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output
yytext	The text of the matched pattern is stored in this variable (char*).
yyleng	Gives the length of the matched pattern.
yyylineno	Provides current line number information. (May or may not be supported by the lexer.)
yylval	Value associated with the token.

LEX FUNCTIONS	
yylex()	The function that starts the analysis. It is automatically generated by Lex.
yywrap()	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing.
yyless(int n)	This function can be used to push back all but first „n‟ characters of the read token.
yymore()	This function tells the lexer to append the next token to the current token.

1.3 Regular Expressions

It is used to describe the pattern. It is widely used to in lex. It uses meta language. The character used in this meta language are part of the standard ASCII character set. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

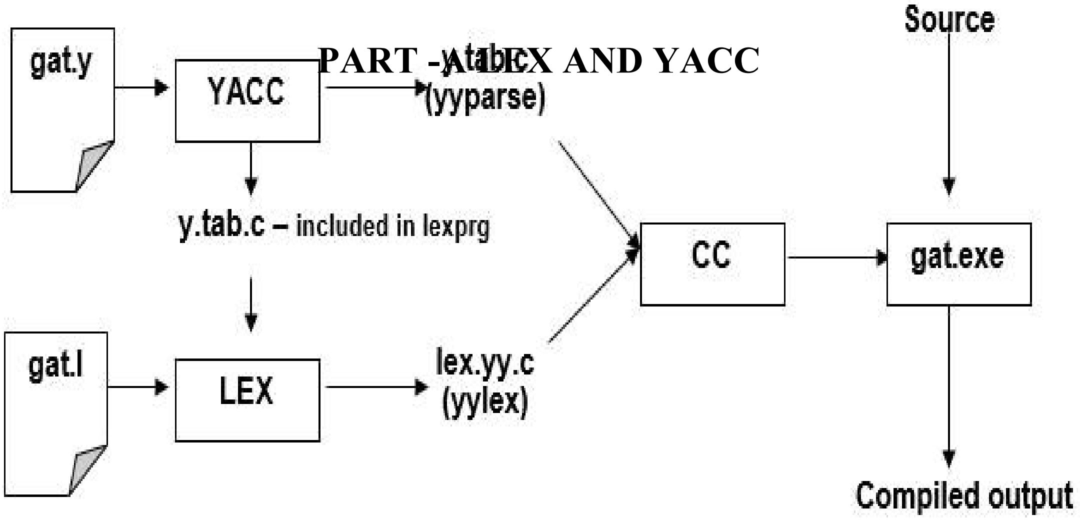
Character	**Meaning**
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches any character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*	Match zero or more occurrences of the preceding pattern.
+	Matches one or more occurrences of the preceding pattern.(no empty string).Ex: [0-9]+ matches “1”,”111” or “123456” but not an empty string.
?	Matches zero or one occurrences of the preceding pattern. Ex:?[0-9]+ matches a signed number including an optional leading minus.
$	Matches end of line as the last character of the pattern.
{ }	1) Indicates how many times a pattern can be present. Example: A{1,3} implies one to three occurrences of A may be present. 2) If they contain name, they refer to a substitution by that name. Ex: {digit}
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table. Ex: \n is a newline character, while “*” is a literal asterisk.
^	Negation.
|	Matches either the preceding regular expression or the following regular expression. Ex: cow|sheep|pig matches any of the three words.
"< symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the Succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions together into a new regular expression. Ex: (01) represents the character sequence 01. Parentheses are useful when building up complex patterns with *,+ and |

2. INTRODUCTION TO YACC

YACC provides a general tool for imposing structure on the input to a computer program. The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. YACC prepares a specification of the input process. YACC generates a function to control the input process. This function is called a parser.

The name is an acronym for "Yet Another Compiler Compiler". YACC generates the code for the parser in the C programming language. YACC was developed at AT& T for the Unix operating system. YACC has also been rewritten for other languages, including Java, Ada.

The function parser calls the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules .The input structure rule is called as grammar. When one of the rule is recognized, then user code supplied for this rule (user code is action) is invoked. Actions have the ability to return values and makes use of the values of other actions.



2.1 Steps in writing YACC Program:

1st Step:	*Using gedit editor create a file with extension y. For example: gedit prg1.y*
2nd Step:	YACC –d prg1.y
3rd Step:	lex prg1.l
4th Step:	cc y.tab.c lex.yy.c -ll
5th Step:	/a.out

When we run YACC, it generates a parser in file y.tab.c and also creates an include file y.tab.h. To obtain tokens, YACC calls yylex. Function yylex has a return type of int, and returns the token. Values associated with the token are returned by lex in variable yylval.

2.2 Structure of YACC source program:

Basic Specification:

Every YACC specification file consists of three sections. The declarations, Rules (of grammars), programs. The sections are separated by double percent "%%" marks. The % is generally used in YACC specification as an escape character.

The general format for the YACC file is very similar to that of the Lex file.

{definitions}
%%
{rules}
%%
{user subroutines}

%% is a delimiter to the mark the beginning of the Rule section.

2.2.1 Definition Section:

%union	It defines the Stack type for the Parser. It is a union of various datas/structures/ Objects
%token	These are the terminals returned by the yylex function to the YACC. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member>tokenName. Ex: %token NAME NUMBER
%type	The type of a non-terminal symbol in the Grammar rule can be specified with this.The format is %type <stack member>non-terminal.
%noassoc	Specifies that there is no associatively of a terminal symbol.
%left	Specifies the left associatively of a Terminal Symbol
%right	Specifies the right associatively of a Terminal Symbol.
%start	Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
%prec	Changes the precedence level associated with a particular rule to that of the following token name or literal

2.2.2 Rules Section:

The rules section simply consists of a list of grammar rules. A grammar rule has the form:

A: BODY

A represents a nonterminal name, the colon and the semicolon are YACC punctuation and BODY represents names and literals. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. The literal consists of a character enclosed in single quotes. Names representing tokens must be declared as follows in the declaration sections:

%token name1 name2…

Every name not defined in the declarations section is assumed to represent a non- terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule. Of all the no terminal symbols, one, called the start symbol has a particular importance. The parser is designed to recognize the start symbol. By default the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

With each grammar rule, the user may associate actions to be. These actions may return values, and may obtain the values returned by the previous actions. Lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement. Actions are enclosed in curly braces.

PART A-LEX PROGRAMS

1a.Program to count the number of characters, words, spaces and lines in a given input file.

Aim:

Algorithm:

Program:

Output:

1b.Program to recognize and count the number of identifiers in a file.

Aim:

Algorithm:

Program:

Output:

2a.Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

Aim:

Algorithm:

Program:

Output:

2b.Program to recognize whether a given sentence is simple or compound.

Aim:

Algorithm:

Program:

Output:

3a.Program to count no of:

i.+ve and –ve integers
ii. +ve and –ve fractions

Aim:

Algorithm:

Program:

Output:

3b.Program to count the no of ‘scanf’ and ‘printf’ statements in a C program. Replace them with ‘readf’ and ‘writef’ statements respectively.

Aim:

Algorithm:

Program:

Output:

PART A-YACC PROGRAMS

4. Program to evaluate arithmetic expression involving operators

+,-,*, / Aim:

Algorithm:

Program:

Output:

5. Program to recognize a valid variable which starts with a letter, followed by any number of letters or digits.

Aim:

Algorithm:

Program:

Output:

6. Program to recognize the strings using the grammar (a^{n}b$^{n;}$

n>=0) Aim:

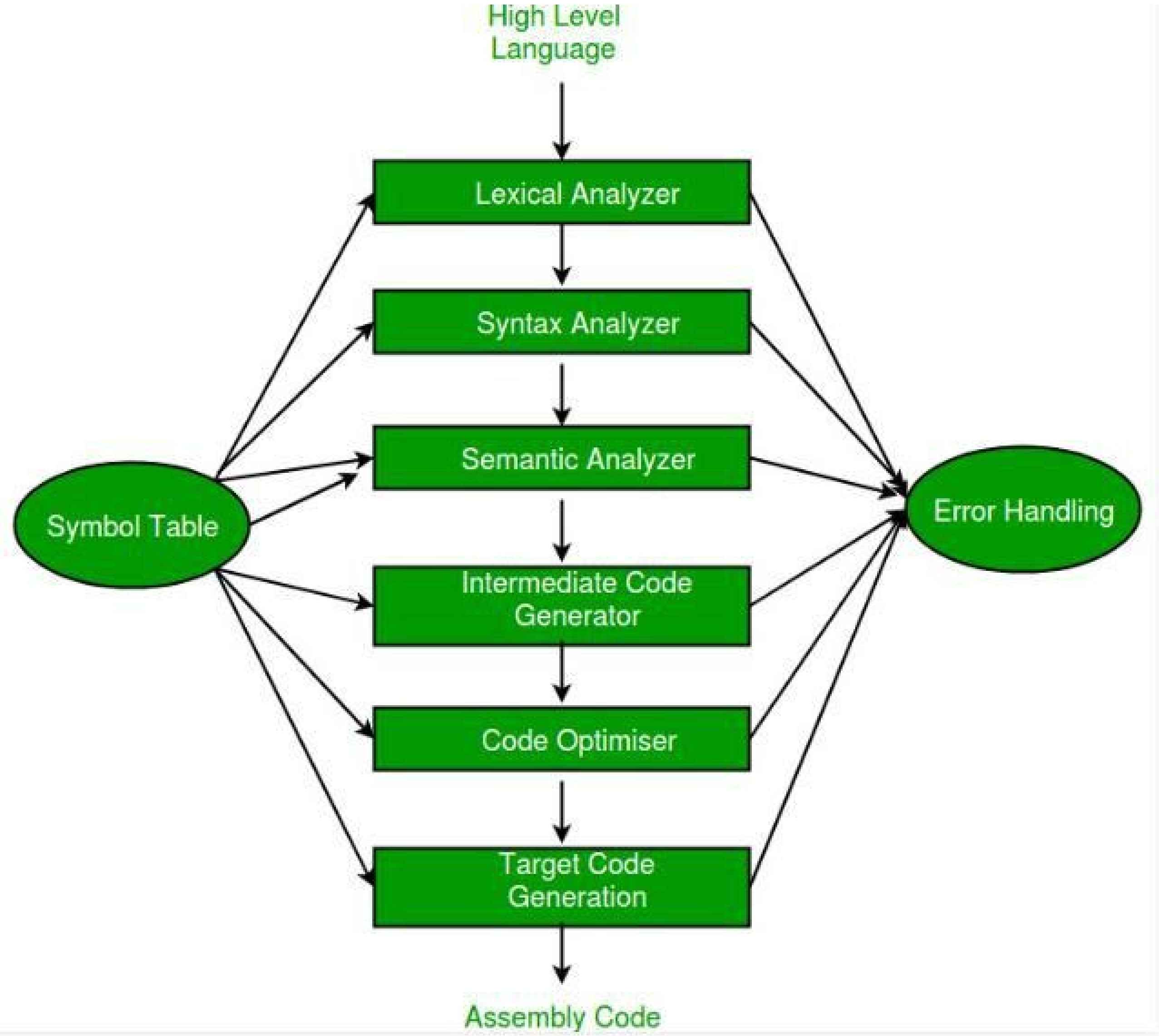
Algorithm:

Program:

Output:

3. INTRODUCTION TO COMPILER DESIGN

Compiler is software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).



Analysis Phase – An intermediate representation is created from the give source code :

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

Lexical analyzer divides the program into "tokens", Syntax analyzer recognizes "sentences" in the program using syntax of language and Semantic analyzer checks static semantics of each construct.

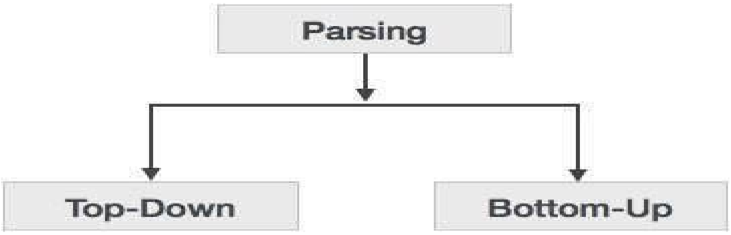
Synthesis Phase – Equivalent target program is created from the intermediate representation. It has three parts :

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

Intermediate Code Generator generates "abstract" code, Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

Syntax analyzers

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.

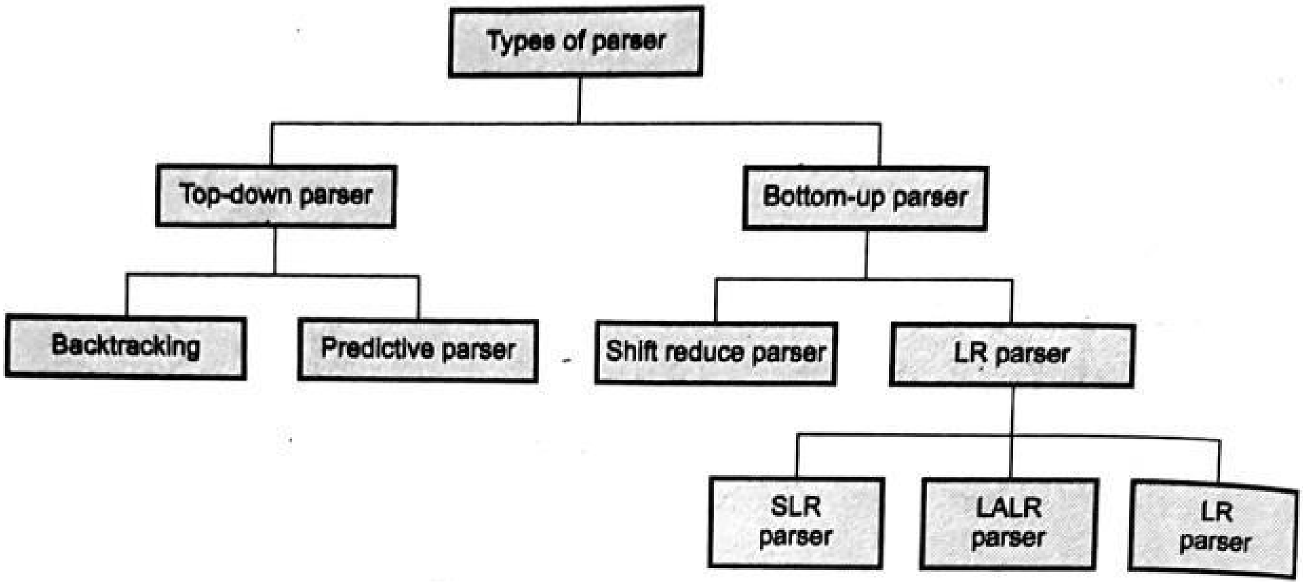


Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

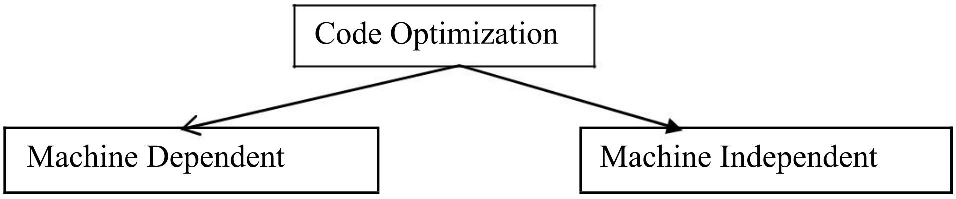
Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.



Code Optimization

Code Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.



In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

PART B- COMPILER DESIGN PROGRAMS

7. C Program to implement Pass1 of Assembler Aim:

Algorithm:

Program:

Output:

8. C Program to implement Absolute

Loader Aim:

Algorithm:

Program:

Output:

8. C program to find the FIRST in context free

grammar. Aim:

Algorithm:

Program:

Output:

10.C Program to implement Shift Reduce Parser for the given grammar

E →E+E
E→E*E
E→(E)
E → id

Aim:

Algorithm:

Program:

Output

11. C Program to implement code optimization

techniques. Aim:

Algorithm:

Program:

Output:

Dayananda Sagar University (DSU) is a premier multi-disciplinary private University launched by Dayananda Sagar Institutions (DSI) in year 2015 in the State of Karnataka. DSU inherits the great legacy of DSI. It offers courses in Engineering, Computer Applications, Sciences, Arts and Management at the Bachelors, Masters and PhD levels. It houses world-class labs spread over 25,000 sq.ft supporting Research & Innovation. Incubation center in newer areas of technology supporting the ICT domains and health care, energy, life science and other developing fields of study is also housed on campus.

Dayananda Sagar University
Hosur Main Road, Kudlu Gate, Hongasandra Village, Begur Hobli, Bangalore, Karnataka, India.
Phone: +91-80-49092924, 49092933, 26662226. Fax: +91-80-26660789.
Email : dean-engg@dsu.edu.in