



**SRM Institute of Science and Technology**

**College of Engineering and Technology**

**School of Computing**

**Department of Computational Intelligence**

**B.Tech – Computer Science and Engineering**

**Regulations 2018**

**18CSC304J – Compiler Design**

**LAB MANUAL**

**Course Coordinator**

**Dr.C.Amuthadevi**

# **SRM Institute of Science and Technology**

## **College of Engineering and Technology**

### **School of Computing**

#### **Department of Computational Intelligence**

#### **B.Tech – Computer Science and Engineering**

#### **Regulations 2018**

### **18CSC304J – Compiler Design**

#### **Department Vision**

To build a world-renowned academic platform in Computational Intelligence by providing unique learning and research experiences in collaboration with industries and world-class universities.

#### **Department Mission**

- To envision in creating, acquiring, and disseminating engineering knowledge on computational intelligence to elevate a student into a professional by imparting knowledge on mathematics, computing sciences, artificial intelligence, and software engineering along with the skills of cognitive computing.
- To offer a unique learning environment through world class faculty, curriculum, modernized lab facilities, and an interactive classroom environment with real-time experience from industrial experts that leads to a computing career in the latest technologies.
- To uplift the innovative research and development in computational intelligence and its allied fields by collaborating with renowned academic institutions and industries.
- To produce graduates who are global innovators and leaders in the development of computational intelligence-based systems, along with the commitment to ethical responsibilities and lifelong learning.

### List of Experiments

S.No	Topic(s)	CO	PO
1	Write a simple calculator program in C/C++/JAVA	1	1
2	Write a program using FLEX.	1	2,5
3	Implementation of scanner by specifying Regular Expressions.	1	2,5
4	Write a program using BISON.	2	2,5
5	Write a program for Top Down Parsing - predictive parsing table (Removal of Left recursion/Left factoring and Compute FIRST & FOLLOW).	2	2
6	Write a program for Bottom Up Parsing - SLR Parsing	3	2
7	Introduction to basic Java - Programs in java	3	2
8	Write a program to traverse syntax trees and perform action arithmetic operations	3	2
9	Write an Intermediate code generation for If/While.	4	3
10	Code Generation Introduction to MIPS Assembly language- (Teach SPIM MIPS simulator).	4	3
11	Write a program to generate machine code for a simple statement	5	3
12	Write a program to generate machine code for an indexed assignment statement	5	3

## **Lab1 : Write a simple calculator program in C/C++/JAVA.**

### **Features of the calculator:**

- Take input from standard input.
- On each line an arithmetic expression can be given in the standard format and the calculator must print the o/p after that.
- The calculator should exit, when the user enters the Ctrl^D (eof) character.
- Supported operators: +, -, \*, /, ^, and ().

### **Algorithm for simple calculator:**

1. Read a line of input
2. If input is Ctrl^D terminate the program
3. If the expression is not balanced for parenthesis report error and go to step 1
4. Convert the expression into post-fix notation
5. Scan through the post-fix expression and push the operands into the stack in the order they appear
6. When any operator is encountered pop the top two stack elements (operands) and execute the operation
7. Push the result on to the stack
8. Print the result and go to step 1

### **Algorithm for post-fix conversion:**

1. Scan the expression from left to right
2. If the scanned character is operand place it in postfix expression
3. If the scanned character is an operand
  - a. If the precedence and associativity of the scanned operator is larger than the one in the stack then push it onto the stack.
  - b. Else pop all the operators from the stack, which are greater than or equal to in precedence and place it in postfix expression and push the scanned operator into the stack. Popping should be stopped if a parenthesis is encountered.
4. If the scanned character is a '(', push it into the stack
5. If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is scanned.
7. Once the scanning is over, pop the stack and add the operators in the postfix expression until it is not empty.
8. Return the postfix expression.

### **Balancing Symbol algorithm:**

1. Traverse the expression from left to right.
  - a. If the current character is a starting bracket, then push it to stack

- b. If the current character is a closing bracket, then pop from stack and if the popped character is not matching starting bracket, return “not balanced”.
2. If there is some starting bracket left in stack then return “not balanced”
3. Return balanced

Example run of the program:

Input:  $2 + 3 * 5$

Output: 17

Input 2:  $(2 + 3) * 5$

Output 2: 25

Input 3:  $30 / 5 / 2$

Output 3: 3

Input 4:  $2 ^ 3 ^ 2$

Output 4: 64

Input 5:  $(2 ^ 3) ^ 2$

Output 5: 36

## Lab 2: Write a program Using FLEX

Flex is a scanner generator tool for lexical analysis, which is based on finite state machine (FSM). The input is a set of regular expressions, and the output is the code to implement the scanner according to the input rules.

To implement a scanner for calculator, we can write the file “cal1.l” as below:

```
/* this is only for scanner, not link with parser yet */
%{
int lineNum = 0;
%}
%%
 "(" { printf("(\\n"); }
 ")" { printf("\\n"); }
 "+" { printf("+\\n"); }
 "*" { printf("*\\n"); }
 \\n { lineNum++; }
 [ \\t]+ { }
 [0-9]+ { printf("%s\\n", yytext); }
%%
int yywrap() {
return 1;
}
int main () {
yylex();
return 0;
}
```

Here is the Makefile used to build the scanner:

```
p1: lex.yy.o
gcc -g -o p1 lex.yy.o
lex.yy.o: cal1.l
flex cal1.l; gcc -g -c lex.yy.c
clean:
rm -f p1 *.o lex.yy.c
```

Simple calculator program in bison

```
//yacc file
#include<stdio.h>;
int regs[26];
int base;
% }
```

```

%start list
%union { int a; }

%token DIGIT LETTER
%left &#39;|&#39;;
%left &#39;&amp;&#39;;
%left &#39;+&#39; &#39;-&#39;;
%left &#39;*&#39; &#39;/&#39; &#39;%&#39;;
%left UMINUS /*supplies precedence for unary minus */
%% /* beginning of rules section */
list: /*empty */
|
list stat &#39;\n&#39;;
|
list error &#39;\n&#39;;
{
yyerrok;
}
;
stat: expr
{
printf(&quot;%d\n&quot;,$1);
}
|
LETTER &#39;=&#39; expr
{
regs[$1.a] = $3.a;
}
;

```

### Lab3 : Implementation of scanner by specifying Regular Expressions

#### Procedure to do the exercise

1. Place the following files in a folder minijava.html, P1.tab.h, P1.l and a sample input file (Factorial.java).
2. flex P1.l
3. gcc lex.yy.c -o P1
4. P1
5. Give = if as input  
Output: = number (number is symbol table id of =)  
if number (number is symbol table id of if)
6. Fill in the code that you have been asked to include in P1.l
7. flex P1.l
8. gcc lex.yy.c -o P1
9. P1 < Factorial.java
10. All tokens generated by the parser will be placed in the output

The content for minijava.html

Goal ::= MainClass ( TypeDeclaration )\* <EOF>

MainClass ::= "class" Identifier "{" "public" "static" "void" "main" "(" "String" "["  
"]" Identifier ")" "{" PrintStatement "}" "}"

TypeDeclaration ::= ClassDeclaration

| ClassExtendsDeclaration

ClassDeclaration ::= "class" Identifier "{" ( VarDeclaration )\* ( MethodDeclaration  
)\* "}"

ClassExtendsDeclaration ::= "class" Identifier "extends" Identifier "{" (  
VarDeclaration )\*

( MethodDeclaration )\* "}"

VarDeclaration ::= Type Identifier ";"

MethodDeclaration ::= "public" Type Identifier "(" ( FormalParameterList )? ")" "{"  
( VarDeclaration )\*

( Statement )\* "return" Expression ";" "}"

FormalParameterList ::= FormalParameter ( FormalParameterRest )\*

FormalParameter ::= Type Identifier

FormalParameterRest ::= "," FormalParameter

Type ::= ArrayType

| BooleanType

| IntegerType

| Identifier

ArrayType ::= "int" "[" "]"

BooleanType ::= "boolean"

IntegerType ::= "int"

Statement ::= Block

| AssignmentStatement



| ArrayAssignmentStatement  
 | IfStatement  
 | WhileStatement  
 | PrintStatement  
 Block ::= "{" ( Statement )\* "  
 AssignmentStatement ::= Identifier "=" Expression ";"  
 ArrayAssignmentStatement ::= Identifier "[" Expression "]" "=" Expression ";"  
 IfStatement ::= "if" "(" Expression ")" Statement "else" Statement  
 WhileStatement ::= "while" "(" Expression ")" Statement  
 PrintStatement ::= "System.out.println" "(" Expression ")" ";"  
 Expression ::= AndExpression  
 | CompareExpression  
 | PlusExpression  
 | MinusExpression  
 | TimesExpression  
 | ArrayLookup  
 | ArrayLength  
 | MessageSend  
 | PrimaryExpression  
 AndExpression ::= PrimaryExpression "&" PrimaryExpression  
 CompareExpression ::= PrimaryExpression "<" PrimaryExpression  
 PlusExpression ::= PrimaryExpression "+" PrimaryExpression  
 MinusExpression ::= PrimaryExpression "-" PrimaryExpression  
 TimesExpression ::= PrimaryExpression "\*" PrimaryExpression  
 ArrayLookup ::= PrimaryExpression "[" PrimaryExpression "]"  
 ArrayLength ::= PrimaryExpression "." "length"  
 MessageSend ::= PrimaryExpression "." Identifier "(" ( ExpressionList )? ")"  
 ExpressionList ::= Expression ( ExpressionRest )  
 ExpressionRest ::= "," Expression  
 PrimaryExpression ::= IntegerLiteral  
 | TrueLiteral  
 | FalseLiteral  
 | Identifier  
 | ThisExpression  
 | ArrayAllocationExpression  
 | AllocationExpression  
 | NotExpression  
 | BracketExpression  
 IntegerLiteral ::= <INTEGER\_LITERAL>  
 TrueLiteral ::= "true"  
 FalseLiteral ::= "false"  
 Identifier ::= <IDENTIFIER>  
 ThisExpression ::= "this"  
 ArrayAllocationExpression ::= "new" "int" "[" Expression "]"

```

AllocationExpression ::= "new" Identifier "(" ")"
NotExpression ::= "!" Expression
BracketExpression ::= "(" Expression ")"
The content for P1.tab.h
/* A Bison parser, made by GNU Bison 2.5. */
/* Bison interface for Yacc-like parsers in C
Copyright (C) 1984, 1989-1990, 2000-2011 Free Software Foundation, Inc.
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>. */
/* As a special exception, you may create a larger work that contains
part or all of the Bison parser skeleton and distribute that work
under terms of your choice, so long as that work isn't itself a
parser generator using the skeleton or a modified version thereof
as a parser skeleton. Alternatively, if you modify or redistribute
the parser skeleton itself, you may (at your option) remove this
special exception, which will cause the skeleton and the resulting
Bison output files to be licensed under the GNU General Public
License without this special exception.
This special exception was added by the Free Software Foundation in
version 2.2 of Bison. */

/* Tokens. */
#ifndef YYTOKENTYPE
#define YYTOKENTYPE
/* Put the tokens into the symbol table, so that GDB and other debuggers
know about them. */
enum yytokentype {
HASHDEFINE = 258,
NOT = 259,
CURLY_OPEN = 260,
CURLY_CLOSE = 261,
PAR_OPEN = 262,
PAR_CLOSE = 263,
SQR_CLOSE = 264,
IF = 265,
WHILE = 266,

```

```

CLASS = 267,
PUBLIC = 268,
STATIC = 269,
VOID = 270,
MAIN = 271,
STR = 272,
PRINTLN = 273,
EXTENDS = 274,
THIS = 275,
NEW = 276,
SEMI_COLON = 277,
COMMA = 278,
LENGTH = 279,
TRUE = 280,
FALSE = 281,
NUMBER = 282,
RET = 283,
BOOL = 285,
INT = 286,
IDENTIFIER = 287,
ADD = 288,
SUB = 289,
MUL = 290,
DIV = 291,
MOD = 292,
BIT_AND = 293,
LESSTHAN = 294,
SQR_OPEN = 295,
DOT = 296,
ASSIGNMENT = 297,
ELSE = 298,
lab1 = 299,
newlabel = 300,
label = 301
};
#endif
#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE
{
/* Line 2068 of yacc.c */
#line 25 "P1.y"
char* str;
/* Line 2068 of yacc.c */
#line 102 "P1.tab.h"

```

```

} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define YYSTYPE YYSSTYPE /* obsolescent; will be withdrawn */
# define YYSSTYPE_IS_DECLARED 1
#endif
extern YYSSTYPE yylval;
Template of the lex (P1.1) file with code for one operator and one keyword.
%{
#include "P1.tab.h"
#include<stdio.h>
#include<stdlib.h>
%}
%%
/* Write code to ignore empty spaces and newlines. */
/* Write code to ignore comments (single line and multiline). */
/* Write code to scan all the operators, paranthesis etc. Example shown for
assignment. */
"=" { char* str=yytext;printf ("%s %d\n", str, ASSIGNMENT);}
/* Write code to scan all the keywords. Example shown for if */
"if" { char* str=yytext;printf ("%s %d\n", str, IF);}

* Write code to scan all the punctuations. */
/* Write code to scan numbers and identifiers. */
/* Write code to throw error if you encounter any invalid token. */
%%
int main(){
yylex();
}
int yywrap(){ }
The content for Factorial.java
class Factorial{
public static void main(String[] a){
System.out.println(new Fac().ComputeFac((10)));
}
}
class Fac { // Test
public int ComputeFac(int num){
int num_aux ;
if (num < 2)
num_aux = 1 ;
else
num_aux = num * (this.ComputeFac(num-1)) ;
return num_aux ;
}}

```

#### Lab 4: Write a program using BISON.

##### Simple calculator program in bison

```
//yacc file
#include<stdio.h>;
int regs[26];
int base;
% }
%start list
%union { int a; }

%token DIGIT LETTER
%left &#39;|&#39;;
%left &#39;&amp;&#39;;
%left &#39;+&#39; &#39;-&#39;;
%left &#39;*&#39; &#39;/&#39; &#39;%&#39;;
%left UMINUS /*supplies precedence for unary minus */
%% /* beginning of rules section */
list: /*empty */
|
list stat &#39;\n&#39;;
|
list error &#39;\n&#39;;
{
yyerrok;
}
;
stat: expr
{
printf(&quot;%d\n&quot;,$1);
}
|
LETTER &#39;=&#39; expr
{
regs[$1.a] = $3.a;
}
;

expr: &#39;(&#39; expr &#39;)&#39;;
{
$$ = $2;
}
|
expr &#39;*&#39; expr
{
```

```

$$$.a = $1.a * $3.a;
}
|
expr &#39;/&#39; expr
{
$$$.a = $1.a / $3.a;
}
|
expr &#39;%&#39; expr
{
$$$.a = $1.a % $3.a;
}
|
expr &#39;+&#39; expr
{
$$$.a = $1.a + $3.a;
}
|
expr &#39;-&#39; expr
{
$$$.a = $1.a - $3.a;
}
|
expr &#39;&amp;&#39; expr
{
$$$.a = $1.a &amp; $3.a;
}
|
expr &#39;|&#39; expr
{
$$$.a = $1.a | $3.a;
}
|
&#39;-&#39; expr %prec UMINUS
{
$$$.a = -$2.a;
}

|
LETTER
{
$$$.a = regs[$1.a];
}
|

```

```

number
;
number: DIGIT
{
$$ = $1;
base = ($1.a==0) ? 8 : 10;
} |
number DIGIT
{
$$ .a = base * $1.a + $2.a;
}
;
%%
main()
{
return(yyparse());
}
yyerror(s)
char *s;
{
fprintf(stderr, "%s\n", s);
}
yywrap()
{
return(1);
}

```

## LEX FILE

```

#include <stdio.h>
#include "y.tab.h"
int c;
% }
%%
" " ;
[a-z] {
c = yytext[0];
yylval.a = c - 'a';
return(LETTER);
}
[0-9] {
c = yytext[0];
yylval.a = c - '0';
return(DIGIT);
}

```

```
}  
[^a-z0-9\b] {  
c = yytext[0];  
return(c);  
}  
%%
```

1) Create the calc.yacc file  
2) Create the calc.lex file  
3) Run the following commands:  
yacc -d calc.yacc  
lex -d calc.lex  
cc y.tab.c lex.yy.c  
./a.out

### **OUTPUT:**

5+4  
9



**5. Write a program for Top Down Parsing - predictive parsing table (Removal of Left recursion/Left factoring and Compute FIRST & FOLLOW).**

**Procedure**

**Algorithm: LeftRecursionFirstFollowParsingTable**

Input: Grammar G

Output: Parsing table

1. Remove left recursion from the grammar G.
  - a. For each non-terminal A in G, do the following:
    - i. If there exists a production  $A \rightarrow A\alpha \mid \beta$ , where  $\beta$  is not starting with A, split it into:  
 $A \rightarrow \beta A'$   
 $A' \rightarrow \alpha A' \mid \epsilon$  (epsilon), where  $\epsilon$  represents the empty string.
    - ii. If  $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$  are the productions of A after step 1, remove  $\beta_i$  if  $\beta_i$  starts with A.
2. Compute FIRST sets for each non-terminal and terminal symbol in the grammar G.
  - a. Initialize FIRST set for each terminal as itself.
  - b. For each non-terminal A in G, initialize FIRST(A) as an empty set.
  - c. Repeat until no changes in FIRST sets:
    - i. For each production  $A \rightarrow \alpha$ , do the following:
      - If  $\alpha$  is terminal or  $\epsilon$ , add  $\alpha$  to FIRST(A).
      - If  $\alpha$  is non-terminal, add all symbols from FIRST( $\alpha$ ) to FIRST(A), except  $\epsilon$ .
      - If  $\epsilon$  is in FIRST( $\alpha$ ), continue to the next symbol.
3. Compute FOLLOW sets for each non-terminal in the grammar G.
  - a. Initialize FOLLOW set for the start symbol S as { \$ }, where \$ is the end marker.
  - b. Repeat until no changes in FOLLOW sets:
    - i. For each production  $A \rightarrow \alpha B \beta$ , where B is a non-terminal:
      - Add all symbols from FIRST( $\beta$ ) to FOLLOW(B), except  $\epsilon$ .
      - If  $\beta$  is  $\epsilon$  or the symbols in  $\beta$  derive  $\epsilon$ , add all symbols from FOLLOW(A) to FOLLOW(B).
4. Construct the parsing table.
  - a. Initialize parsing table M with empty entries.
  - b. For each production  $A \rightarrow \alpha$  in G, do the following:

- i. For each terminal  $a$  in  $FIRST(\alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$ .
  - If  $\alpha$  derives  $\epsilon$ , add  $A \rightarrow \alpha$  to  $M[A, b]$  for each terminal  $b$  in  $FOLLOW(A)$ .
- ii. If  $\epsilon$  is in  $FIRST(\alpha)$ , for each terminal  $b$  in  $FOLLOW(A)$ , add  $A \rightarrow \alpha$  to  $M[A, b]$ .

5. Return the parsing table  $M$ .

### **Remove left recursion from the grammar $G$**

```
public static List<String> removeLeftRecursion(List<String> grammar) {
    List<String> newGrammar = new ArrayList<>();

    // Iterate over each non-terminal in the grammar
    for (String production : grammar) {
        String[] parts = production.split("->");
        String nonTerminal = parts[0].trim();
        String[] alternatives = parts[1].trim().split("\\|");

        List<String> withRecursion = new ArrayList<>();
        List<String> withoutRecursion = new ArrayList<>();

        // Separate productions with and without left recursion
        for (String alt : alternatives) {
            if (alt.trim().startsWith(nonTerminal)) {
                withRecursion.add(alt.trim().substring(nonTerminal.length()).trim() + " " + nonTerminal + "");
            } else {
                withoutRecursion.add(alt.trim() + " " + nonTerminal + "");
            }
        }
    }
}
```

### **Removal of left factor**

```
public static List<String> leftFactor(List<String> grammar) {
    List<String> newGrammar = new ArrayList<>();

    // Iterate over each non-terminal in the grammar
    for (String production : grammar) {
        String[] parts = production.split("->");
        String nonTerminal = parts[0].trim();
        String[] alternatives = parts[1].trim().split("\\|");

        // Find the common prefix among alternatives
        String commonPrefix = findCommonPrefix(alternatives);
```

```

// If there is a common prefix, perform left factoring
if (!commonPrefix.isEmpty()) {
    StringBuilder sb = new StringBuilder();
    sb.append(nonTerminal).append(" -> ").append(commonPrefix).append("
").append(nonTerminal).append("");
    newGrammar.add(sb.toString());

    StringBuilder sb2 = new StringBuilder();
    for (String alt : alternatives) {
        if (alt.startsWith(commonPrefix)) {
            sb2.append(alt.substring(commonPrefix.length()).trim());
            sb2.append(" | ");
        }
    }
    sb2.append("ε");
    newGrammar.add(nonTerminal + " -> " + sb2.toString());
} else {
    // If no common prefix, keep the production as is
    newGrammar.add(production);
}
}

return newGrammar;
}

```

### **Parse table creation**

```

public class TopDownParser {
    public static void main(String[] args) {
        // Example grammar: S -> A, A -> aA | ε
        Map<String, Set<String>> firstSets = new HashMap<>();
        Map<String, Set<String>> followSets = new HashMap<>();

        // Construct FIRST sets
        firstSets.put("S", new HashSet<>());
        firstSets.put("A", new HashSet<>());

        firstSets.get("S").add("a");
        firstSets.get("A").add("a");
        firstSets.get("A").add("ε");

        // Construct FOLLOW sets
        followSets.put("S", new HashSet<>());
        followSets.put("A", new HashSet<>());

        followSets.get("S").add("$");
    }
}

```

```

followSets.get("A").add("$");
followSets.get("A").add("a");

// Construct parse table
Map<String, Map<String, String>> parseTable = new HashMap<>();

parseTable.put("S", new HashMap<>());
parseTable.put("A", new HashMap<>());

parseTable.get("S").put("a", "A");
parseTable.get("A").put("a", "aA");
parseTable.get("A").put("$", "ε");

// Print FIRST sets
System.out.println("FIRST sets:");
for (Map.Entry<String, Set<String>> entry : firstSets.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// Print FOLLOW sets
System.out.println("\nFOLLOW sets:");
for (Map.Entry<String, Set<String>> entry : followSets.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// Print parse table
System.out.println("\nParse Table:");
for (Map.Entry<String, Map<String, String>> entry : parseTable.entrySet()) {
    String nonTerminal = entry.getKey();
    Map<String, String> actions = entry.getValue();
    System.out.println("Non-Terminal: " + nonTerminal);
    System.out.println("Actions: " + actions);
}
}
}

```

## EXPRESSION GRAMMAR

E->E+T

E->T

T->T\*F

T->F

F->( E)

F->id

## OUTPUT

### PARSE TABLE

	)	+	*	(	i	d
E				E->TE'	E->TE'	
T				T->FT'	T->FT'	
F				F->(E)	F->i	
E'	E'->e	E'->TE'				
T'	T'->e	T'->e	T'->FT'			

## **Lab 6: Write a program for Bottom Up Parsing - SLR Parsing**

### **PROCEDURE:**

1. Create a parsing table: Construct a parsing table that maps a pair of stack-top symbol and input symbol to a parsing action. The table will tell us whether to shift, reduce or accept the input.
2. Initialize the stack: Initialize the stack with the start symbol and an end-of-input symbol (\$). We begin parsing with the input buffer (a sequence of tokens) on the left and the stack on the right.
3. Shift or reduce: The parsing algorithm will shift the next input symbol onto the stack or reduce the top of the stack if possible, according to the parsing table. If the action is shift, the parser will push the input symbol onto the stack. If the action is reduce, the parser will replace the top of the stack with the non-terminal on the right-hand side of the production rule, and repeat the process until it can no longer reduce.
4. Handle conflicts: In some cases, the parsing table may contain conflicts that prevent the parser from making a decision. Conflicts occur when a cell in the parsing table has multiple entries. There are two types of conflicts - shift-reduce and reduce-reduce conflicts. To resolve these conflicts, the parser will use a set of rules to determine which action to take.
5. Accept or reject: After parsing is complete, if the stack contains only the end-of-input symbol and the start symbol, the input is accepted. Otherwise, the input is rejected.

**PROGRAM:**

```
import java.util.*;

class SLRParser {
    private final List<String> grammar;
    private final Map<Integer, Map<String, String>> actionTable;
    private final Map<Integer, Map<String, Integer>> goToTable;
    private final Stack<Integer> stateStack;
    private final Stack<String> symbolStack;
    private final List<String> input;

    public SLRParser(List<String> grammar, Map<Integer, Map<String, String>>
actionTable, Map<Integer, Map<String, Integer>> goToTable, List<String> input)
    {
        this.grammar = grammar;
        this.actionTable = actionTable;
        this.goToTable = goToTable;
        this.input = input;
        this.stateStack = new Stack<>();
        this.symbolStack = new Stack<>();
    }

    public boolean parse() {
        stateStack.push(0);
        int inputIndex = 0;

        while (inputIndex < input.size()) {
            int currentState = stateStack.peek();
            String currentSymbol = input.get(inputIndex);

            if (actionTable.containsKey(currentState) &&
actionTable.get(currentState).containsKey(currentSymbol)) {
                String action = actionTable.get(currentState).get(currentSymbol);

                if (action.startsWith("s")) {
                    int nextState = Integer.parseInt(action.substring(1));
                    stateStack.push(nextState);
                    symbolStack.push(currentSymbol);
                    inputIndex++;
                } else if (action.startsWith("r")) {
                    int productionIndex = Integer.parseInt(action.substring(1));
                    String production = grammar.get(productionIndex);
                    String[] parts = production.split("->");
                    String leftPart = parts[0].trim();
```

```

        String rightPart = parts[1].trim();
        int len = rightPart.split(" ").length;
        for (int i = 0; i < len; i++) {
            stateStack.pop();
            symbolStack.pop();
        }
        currentState = stateStack.peek();
        String nonTerminal = leftPart;
        stateStack.push(goToTable.get(currentState).get(nonTerminal));
        symbolStack.push(nonTerminal);
    } else if (action.equals("accept")) {
        return true;
    }
} else {
    // No valid action in action table, parsing fails
    return false;
}
}

// No more input symbols, parsing fails
return false;
}
}

public class Main {
    public static void main(String[] args) {
        // Example grammar
        List<String> grammar = Arrays.asList(
            "S -> E",
            "E -> E + T",
            "E -> T",
            "T -> id"
        );

        // Example action table
        Map<Integer, Map<String, String>> actionTable = new HashMap<>();
        Map<String, String> action0 = new HashMap<>();
        action0.put("id", "s3");
        actionTable.put(0, action0);
        Map<String, String> action1 = new HashMap<>();
        action1.put("+", "s4");
        actionTable.put(1, action1);
        Map<String, String> action3 = new HashMap<>();
        action3.put("$", "accept");
    }
}

```



```

actionTable.put(3, action3);
Map<String, String> action4 = new HashMap<>();
action4.put("id", "s3");
actionTable.put(4, action4);

// Example go-to table
Map<Integer, Map<String, Integer>> goToTable = new HashMap<>();
Map<String, Integer> goTo0 = new HashMap<>();
goTo0.put("E", 1);
goToTable.put(0, goTo0);
Map<String, Integer> goTo1 = new HashMap<>();
goTo1.put("T", 2);
goToTable.put(1, goTo1);

// Example input
List<String> input = Arrays.asList("id", "+", "id", "$");

SLRParser parser = new SLRParser(grammar, actionTable, goToTable, input);
boolean result = parser.parse();
if (result) {
    System.out.println("Input string accepted by grammar.");
} else {
    System.out.println("Input string rejected by grammar.");
}
}
}

```

OUTPUT:

Enter input word: id\*id

The string is accepted

## Lab 7. Introduction to basic Java - Programs in java

**Introduction to JAVA- Tutorial covers basics of the following  
Java Virtual Machine(JVM), Bytecode in the Development Process, Java Runtime Environment**

### Simple Program

```
import java.util.Scanner;

public class SimpleJavaProgram {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter the first number
        System.out.print("Enter the first number: ");
        int num1 = scanner.nextInt();

        // Prompt the user to enter the second number
        System.out.print("Enter the second number: ");
        int num2 = scanner.nextInt();

        // Compute the sum of the two numbers
        int sum = num1 + num2;

        // Display the result
        System.out.println("The sum of " + num1 + " and " + num2 + " is: " + sum);

        // Close the scanner
        scanner.close();
    }
}
```

**Lab 8. Write a program to traverse syntax trees and perform action arithmetic operations.**

Basic algorithm for traversing syntax trees and performing arithmetic operations

Algorithm: TraverseSyntaxTreeAndPerformArithmetic

Input: Syntax tree T representing an arithmetic expression

1. Define a recursive function Traverse(node), where node is a node in the syntax tree:
  - a. If node is a leaf node containing a number:
    - i. Return the value of the number.
  - b. If node is an interior node representing an operator:
    - i. Let leftValue = Traverse(leftChild), where leftChild is the left child of node.
    - ii. Let rightValue = Traverse(rightChild), where rightChild is the right child of node.
    - iii. Perform the arithmetic operation indicated by the operator stored in node, using leftValue and rightValue.
    - iv. Return the result of the operation.
2. Call Traverse(root), where root is the root node of the syntax tree.

```
class TreeNode {
    String value;
    TreeNode left;
    TreeNode right;

    TreeNode(String value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

public class SyntaxTreeTraversal {
    public static int traverseAndCalculate(TreeNode node) {
        if (node == null) {
            return 0;
        }
    }
}
```

```

    if (node.left == null && node.right == null) {
        return Integer.parseInt(node.value);
    }

    int leftValue = traverseAndCalculate(node.left);
    int rightValue = traverseAndCalculate(node.right);

    switch (node.value) {
        case "+":
            return leftValue + rightValue;
        case "-":
            return leftValue - rightValue;
        case "*":
            return leftValue * rightValue;
        case "/":
            if (rightValue == 0) {
                throw new ArithmeticException("Division by zero");
            }
            return leftValue / rightValue;
        default:
            throw new IllegalArgumentException("Invalid operator: " + node.value);
    }
}

public static void main(String[] args) {
    // Example syntax tree: 3 + (5 * 2)
    TreeNode root = new TreeNode("+");
    root.left = new TreeNode("3");
    root.right = new TreeNode("*");
    root.right.left = new TreeNode("5");
    root.right.right = new TreeNode("2");

    // Perform traversal and calculation
    int result = traverseAndCalculate(root);
    System.out.println("Result: " + result);
}

```

Implemented a method `traverseAndCalculate` to recursively traverse the syntax tree and perform arithmetic operations. In the main method, created an example syntax tree representing the expression  $3 + (5 * 2)$ , traverse the tree, and print the result.

## Lab 9. Write an Intermediate code generation for If/While.

Create a method takes two strings as input: condition represents the condition of the if statement, and action represents the action to be performed if the condition is true.

- Inside the generateIfCode method, a StringBuilder is used to construct the code.
- The condition and action strings are concatenated to form the if statement block.
- Finally, the generated code is returned as a string.

can customize this code by providing different conditions and actions to generate code for different if statements. Additionally, can extend this algorithm to handle more complex if-else or nested if statements as needed.

```
import java.util.*;

public class CodeGenerationForIf {

    public static void main(String[] args) {

        // Example input: if (x > 5) { y = x * 2; }

        String condition = "x > 5";

        String action = "y = x * 2;";

        // Generate code for the if statement

        String generatedCode = generateIfCode(condition, action);

        System.out.println("Generated code:");

        System.out.println(generatedCode);

    }

    public static String generateIfCode(String condition, String action) {

        StringBuilder codeBuilder = new StringBuilder();

        // Add the if statement
```

```
codeBuilder.append("if (").append(condition).append(") {\n");

// Add the action inside the if block

codeBuilder.append("\t").append(action).append("\n");

// Close the if block

codeBuilder.append("}");

return codeBuilder.toString();

}

}
```

**OUTPUT:**

Generated code:

```
if (x > 5) {
    y = x * 2;
}
```

## **Lab 10: Code Generation Introduction to MIPS Assembly language-** (Teach SPIM MIPS simulator).

To learn the basics of MIPS Assembly Language

### **Introduction:**

- MIPS (Million Instructions Per Second) is a RISC (Reduced Instruction Set Computer) instruction set architecture developed by MIPS Technologies in the late 1980s.
- It is a loadstore architecture, meaning that data must be loaded into registers before it can be processed. MIPS is known for its simplicity, efficiency, and high performance.
- It has been widely used in embedded systems, networking equipment, and other applications where performance and power consumption are critical.

### **Algorithm:**

Step 1: get value for register \$a0

Step 2: get value for register \$a1

Step 3: add values in \$a0 and \$a1

Step 4: store result in register \$v0

Step 5: print value in register \$v0

### **Sample Code:**

```
.data
num1: .word 10
num2: .word 5
.text
main:
    # load num1 into $a0
    lw $a0, num1
    # load num2 into $a1
    lw $a1, num2
    # add num1 and num2
    add $v0, $a0, $a1
    # print result
    li $v0, 1 # syscall code for print_int
    syscall
    # exit program
    li $v0, 10 # syscall code for exit
    syscall
```

**Output: 15**

### **How To Run:**

- Open MARS MIPS Simulator.
- Click "File" -> "Open" and select your MIPS assembly file.
- Click "Assemble" to assemble the file.
- Click "Run" to execute the program.

## Lab11: Write a program to generate machine code for a simple statement

To implement a java program to convert an assembly language statement to machine code

### Algorithm:

Step 1: create a byte array to store the machine code

Step 2: set the opcode for the MIPS instruction in the byte array

Step 3: set the source register for the first operand in the byte array

Step 4: set the source register for the second operand in the byte array

Step 5: set the destination register in the byte array

Step 6: set the function code for the MIPS instruction in the byte array

Step 7: write the machine code to a file

### Sample Code:

```
import java.io.FileOutputStream;
import java.io.IOException;
public class GenerateMachineCode {
    public static void main(String[] args) throws IOException {
        // Create a byte array to store the machine code
        byte[] machineCode = new byte[4];
        // Set the opcode for the MIPS instruction in the byte array
        machineCode[0] = (byte) 0x0;
        // Set the source register for the first operand in the byte array
        machineCode[1] = (byte) 0x0;
        // Set the source register for the second operand in the byte array
        machineCode[2] = (byte) 0x80;
        // Set the destination register in the byte array
        machineCode[3] = (byte) 0x21;
        // Write the machine code to a file
        try (FileOutputStream fos = new FileOutputStream("machine_code.bin")){
            fos.write(machineCode);
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

### Explanation:

The machine code instructions for MIPS are 32 bits long. Each instruction is divided into fields, which specify the opcode, source registers, destination register, and function code.



- The opcode for the add instruction is 000000. This is stored in the first byte of the machine code instruction, which is machineCode[0].
- The source registers for the add instruction are \$a0 and \$a1. These are stored in the second and third bytes of the machine code instruction, which are machineCode[1] and machineCode[2].
- The destination register for the add instruction is \$v0. This is stored in the fourth byte of the machine code instruction, which is machineCode[3].
- The function code for the add instruction is 000001. This is also stored in the fourth byte of the machine code instruction, which is machineCode[3].
- Therefore, the machine code instruction 000000000000000010000001 translates to the MIPS assembly instruction add \$v0, \$a0, \$a1.

Here is a breakdown of the machine code instruction:

**000000: Opcode for the add instruction**

**00000: Source register for the first operand (\$a0)**

**00001: Source register for the second operand (\$a1)**

**10000: Destination register (\$v0)**

**000001: Function code for the add instruction**

Assembly Statement: add \$v0, \$a0, \$a1

Machine Statement: 00000000000000000000000010000001

### **How to run:**

```
javac GenerateMachineCode.java
java GenerateMachineCode
```

## Lab 12: Write a program to generate machine code for an indexed assignment statement

### Algorithm:

- Step 1: create a byte array to store the machine code
- Step 2: set the opcode for the MIPS instruction in the byte array
- Step 3: set the source register in the byte array
- Step 4: set the base register in the byte array
- Step 5: set the offset in the byte array
- Step 6: write the machine code to a file

### Sample Code:

```
import java.io.FileOutputStream;
import java.io.IOException;
public class GenerateMachineCodeIndexedAssignment {
    public static void main(String[] args) throws IOException {
        // Create a byte array to store the machine code
        byte[] machineCode = new byte[6];
        // Set the opcode for the MIPS instruction in the byte array
        machineCode[0] = (byte) 0x2b;
        // Set the source register in the byte array
        machineCode[1] = (byte) 0x04;
        // Set the base register in the byte array
        machineCode[2] = (byte) 0x00;
        // Set the offset in the byte array
        machineCode[3] = (byte) 0x00;
        machineCode[4] = (byte) 0x00;
        machineCode[5] = (byte) 0x18;
        // Write the machine code to a file
        try (FileOutputStream fos = new FileOutputStream("machine_code.bin")) {
            fos.write(machineCode);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

### Explanation:

The first 6 bits (101011) are the opcode for the sw instruction.

The next 5 bits (00000) are the source register (\$v0).

The next 5 bits (01000) are the base register (\$a0).

The next 16 bits (00000000000000100) are the offset (4).

Assembly Statement: sw \$v0, 4(\$a0)

Machine Statement: 101011010000010000000000000011000