Name: Saloni Vishwakarma

Roll No. 13

Practical No. 4

Topic: Parser Construction

Platform: Windows or Linux

Language to be used: C, Python or Java (Choose based on the companies targeted for placement)

Aim: Implement SLR Parser.

Theory:

An SLR (Simple LR) parser is a type of bottom-up parser commonly used in compiler construction to analyze the syntax of a programming language. It operates by shifting input symbols onto a stack until it can reduce them to grammar productions, ultimately recognizing whether the input string conforms to the grammar rules of the language.

Here's a simplified overview of how an SLR parser works:

1. Input: The parser takes an input string (sequence of tokens) from the lexical analyser.

2. Stack: It maintains a stack to store grammar symbols and parser states.

3. Parsing Table: The SLR parser uses a parsing table constructed from the grammar to determine its actions. This table typically consists of rows representing states and columns representing input symbols. The entries in the table specify whether to shift, reduce, or accept based on the current state and the next input symbol.

4. Shift Operation: If the parsing table indicates a "shift" operation for the current state and input symbol, the parser shifts the input symbol onto the stack and transitions to a new state.

5. Reduce Operation: If the parsing table indicates a "reduce" operation, the parser pops grammar symbols from the stack according to the right-hand side of a production rule and replaces them with the corresponding non-terminal symbol. This step aims to recognize higher-level constructs in the input.

6. Acceptance: If the parsing table indicates "accept" for a certain state and the end-of-input symbol, the parser successfully recognizes the input string as valid according to the grammar.

SLR parsers are relatively simple and efficient, but they require a grammar that adheres to certain restrictions, such as being in the SLR(1) grammar class. While SLR parsers may not handle all grammars efficiently, they serve as educational tools for understanding parsing algorithms and are often used in introductory compiler courses.

B.

Example-1: Generate LR(0) Parser: Canonical collections of LR(0) items

S → AA
A → aA | b
Solution:
Step: 1 – Grammar Augmentation
S' → .S … Rule 0
S → .AA … Rule 1
A → .aA …. Rule 2
A → .b ….Rule 3
Step: 2 – Closure operation = I0
S' → .S
S → .AA
A → .aA
A→.b

Goto (I0, S) = I1
S' → S. //**
Goto(I0, A) = I2
S → A.A
A → .aA
A → .b
Goto(I0, a) = I3
A → a.A
A→ .aA
A→ .b
Goto(I0,b) = I4
A → b. //**
Goto(I2, A) = I5
S → AA.
Goto (I2, a) = I3

Goto (I2, b) = I4
Goto (I3, A) = I6
A → aA.
Goto (I3, a) = I3
Goto (I3, b) = I4

Rules for construction of parsing table from Canonical collections of LR(0) items

• Action part: For Terminal Symbols

- If A → α.aβ is state Ix in Items and **goto(Ix,a)=Iy** then set action [Ix,a]=Sy (represented as shift to state Iy]
- If A→α. is in Ix, then set **action[Ix,f]** to reduce **A→α** for all symbols "f" where "f" is in Follow(A) (Use rule number)
- If S'→S. is in Ix then set action[Ix,$]=accept.

• Go To Part: For Non Terminal Symbols

- If **goto(Ix, A) = Iy**, then goto(Ix,A) in table **= Y**
- It is numeric value of state Y.
- All other entries are considered as error.
- Initial state is S'→.S

Code

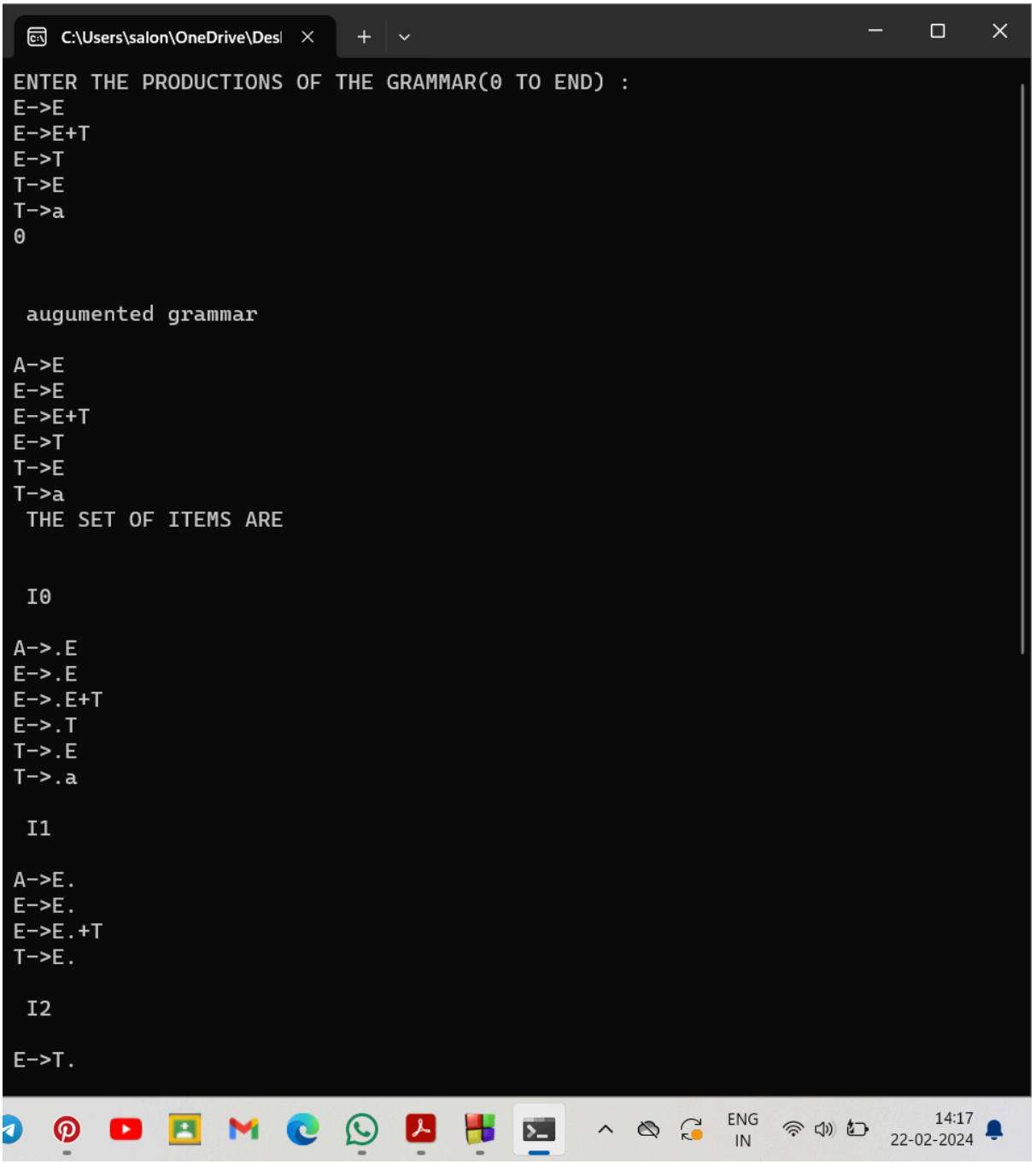
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char prod[20][20], listofvar[26] = "ABCDEFGHIJKLMNOPQR";
int novar = 1, i = 0, j = 0, k = 0, n = 0, m = 0, arr[30];
int noitem = 0;
struct Grammar
{
   char lhs;
   char rhs[8];
```

```
} g[20], item[20], clos[20][10];
int isvariable(char variable)
{
   for (int i = 0; i < novar; i++)
      if (g[i].lhs == variable)
         return i + 1;
   return 0;
}
void findclosure(int z, char a)
{
   int n = 0, i = 0, j = 0, k = 0, l = 0;
   for (i = 0; i < arr[z]; i++)
   {
      for (j = 0; j < strlen(clos[z][i].rhs); j++)
      {
         if (clos[z][i].rhs[j] == '.' && clos[z][i].rhs[j + 1] == a)
         {
            clos[noitem][n].lhs = clos[z][i].lhs;
            strcpy(clos[noitem][n].rhs, clos[z][i].rhs);
            char temp = clos[noitem][n].rhs[j];
            clos[noitem][n].rhs[j] = clos[noitem][n].rhs[j + 1];
            clos[noitem][n].rhs[j + 1] = temp;
            n = n + 1;
         }
      }
   }
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < strlen(clos[noitem][i].rhs); j++)
      {
         if (clos[noitem][i].rhs[j] == '.' && isvariable(clos[noitem][i].rhs[j + 1]) > 0)
         {
            for (k = 0; k < novar; k++)
            {
               if (clos[noitem][i].rhs[j + 1] == clos[0][k].lhs)
               {
                  for (l = 0; l < n; l++)
                     if (clos[noitem][l].lhs == clos[0][k].lhs && strcmp(clos[noitem][l].rhs,
                                                          clos[0][k].rhs) == 0)
                        break;
                  if (l == n)
                  {
                     clos[noitem][n].lhs = clos[0][k].lhs;
                     strcpy(clos[noitem][n].rhs, clos[0][k].rhs);
                     n = n + 1;
                  }
               }
            }
         }
      }
   }
   arr[noitem] = n;
```

```
    int flag = 0;
    for (i = 0; i < noitem; i++)
    {
        if (arr[i] == n)
        {
            for (j = 0; j < arr[i]; j++)
            {
                int c = 0;
                for (k = 0; k < arr[i]; k++)
                    if (clos[noitem][k].lhs == clos[i][k].lhs && strcmp(clos[noitem][k].rhs,
                                                                clos[i][k].rhs) == 0)
                        c = c + 1;
                if (c == arr[i])
                {
                    flag = 1;
                    goto exit;
                }
            }
        }
    }
exit:;
    if (flag == 0)
        arr[noitem++] = n;
}
int main()
{
    printf("ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n");
    do
    {
        scanf("%s", prod[i++]);
    } while (strcmp(prod[i - 1], "0") != 0);
    for (n = 0; n < i - 1; n++)
    {
        m = 0;
        j = novar;
        g[novar++].lhs = prod[n][0];
        for (k = 3; k < strlen(prod[n]); k++)
        {
            if (prod[n][k] != '|')
                g[j].rhs[m++] = prod[n][k];
            if (prod[n][k] == '|')
            {
                g[j].rhs[m] = '\0';
                m = 0;
                j = novar;
                g[novar++].lhs = prod[n][0];
            }
        }
    }
    for (i = 0; i < 26; i++)
        if (!isvariable(listofvar[i]))
            break;
```

```
    g[0].lhs = listofvar[i];
    char temp[2] = {g[1].lhs, '\0'};
    strcat(g[0].rhs, temp);
    printf("\n\n augumented grammar \n");
    for (i = 0; i < novar; i++)
       printf("\n%c->%s ", g[i].lhs, g[i].rhs);
    for (i = 0; i < novar; i++)
    {
       clos[noitem][i].lhs = g[i].lhs;
       strcpy(clos[noitem][i].rhs, g[i].rhs);
       if (strcmp(clos[noitem][i].rhs, "Îµ") == 0)
          strcpy(clos[noitem][i].rhs, ".");
       else
       {
          for (int j = strlen(clos[noitem][i].rhs) + 1; j >= 0; j--)
             clos[noitem][i].rhs[j] = clos[noitem][i].rhs[j - 1];
          clos[noitem][i].rhs[0] = '.';
       }
    }
    arr[noitem++] = novar;
    for (int z = 0; z < noitem; z++)
    {
       char list[10];
       int l = 0;
       for (j = 0; j < arr[z]; j++)
       {
          for (k = 0; k < strlen(clos[z][j].rhs) - 1; k++)
          {
             if (clos[z][j].rhs[k] == '.')
             {
                for (m = 0; m < l; m++)
                   if (list[m] == clos[z][j].rhs[k + 1])
                      break;
                if (m == l)
                   list[l++] = clos[z][j].rhs[k + 1];
             }
          }
       }
       for (int x = 0; x < l; x++)
          findclosure(z, list[x]);
    }
    printf("\n THE SET OF ITEMS ARE \n\n");
    for (int z = 0; z < noitem; z++)
    {
       printf("\n I%d\n\n", z);
       for (j = 0; j < arr[z]; j++)
          printf("%c->%s\n", clos[z][j].lhs, clos[z][j].rhs);
    }
    return 0;
}
```

Output Screenshot



```
 I3

T->a.

 I4

E->E+.T
T->.E
T->.a
E->.E
E->.E+T
E->.T

 I5

E->E+T.
E->T.

 I6

T->E.
E->E.
E->E.+T

Process returned 0 (0x0)   execution time : 37.752 s
Press any key to continue.
```