Compiler Design Lab
CSS651

Group – 2
Members

Saharsh Ananta Jaiswal – 18CS8061
Pudi Pavan Kumar – 18CS8062
Ayesha Uzma – 18CS8063
Rithik Sureka – 18CS8064
Ravupalli Harsha Vardhan – 18CS8065

Assignment – 5

Implement one LR(0) parser. The grammar for the LR(0) parser is another input along with the input text.

Software Used: Google Colab

Theory :

An LR (0) item is a production G with dot at some position on the right side of the production. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing. In the LR (0), we place the reduce node in the entire row.

LR(0) Parse Algorithm :-

1. Initialize the stack with the start state.

2. Read an input symbol

3. while true do

    3.1 Using the top of the stack and the input symbol determine the next state.

    3.2 If the next state is a stack state then

        3.2.1 stack the state

        3.2.2 get the next input symbol

    3.3 else if the next state is a reduce state then

        3.3.1 output reduction number, k

        3.3.2 pop RHSk -1 states from the stack where RHSk is the right hand side of production k.

        3.3.3 set the next input symbol to the LHSk

    3.4 else if the next state is an accept state then

        3.4.1 output valid sentence

        3.4.2 return else

        3.4.3 output invalid sentence

        3.4.4 return

Code:-

```
#---------------- Importing all the Necessary Libraries ----------------

import os
import time
from collections import Counter
!pip install pyfiglet
!pip install termtables
import pyfiglet
import termtables as tt




#---------------- Done Importing Libraries ----------------

# title = pyfiglet.figlet_format("LR (0) Parsing", font="digital")
# print(title)


def addDot(dot):
    addDotVar = dot.replace("->", "->.")
    return addDotVar
```

```
#Function to find closure
def findClosure(gram):
    flag = [gram]
    for i in flag:
        j = i[i.index(".") + 1]
        if j != len(i) - 1:
            for k in productionRules:
                if k[0][0] == j and (addDot(k)) not in flag:
                    flag.append(addDot(k))
        else:
            for k in productionRules:
                if k[0][0] == j and i not in flag:
                    flag.append(i)

    return flag

#---------------- Implementing Functions ----------------
def swapValues(newValue, posValue):
    newValue = list(newValue)
    temp = newValue[posValue]
    if posValue != len(newValue):
        newValue[posValue] = newValue[posValue + 1]
        newValue[posValue + 1] = temp
        newFinal = "".join(newValue)
        return newFinal
    else:
        return "".join(newValue)
```

```
def gotoFucntion(var1):
    arr = []
    pos = var1.index(".")
    if pos != len(var1) - 1:
        j = list(var1)
        k = swapValues(j, pos)
        if k.index(".") != len(k) - 1:
            l = findClosure(k)
            return l
        else:
            arr.append(k)
            return arr
    else:
        return var1

def Terminals(inputTerminal):
    terminalSet = set()
    for p in inputTerminal:
        x1 = p.split('->')
        for t in x1[1].strip():
            if not t.isupper() and t != '.' and t != '':
                terminalSet.add(t)

    terminalSet.add('$')

    return terminalSet

def nonTerminals(gram):
    terms = set()
    for p in gram:
        x1 = p.split('->')
        for t in x1[1].strip():
            if t.isupper():
                terms.add(t)
    return terms

def getList(graph, state):
    finalList = []
    for g in graph:
        if int(g.split()[0]) == state:
            finalList.append(g)

    return finalList

#---------------- Done Implementing Functions  ----------------

#---------------- Augmented Grammar ----------------
productionRules = []
itemSet = []
flag = []
```

```
with open("input.txt", 'r') as fp:
    for i in fp.readlines():
        productionRules.append(i.strip())

productionRules.insert(0, "X->.S")
print("----------------------------------------------------------------")
print("Augmented Grammar")
print(productionRules)
time.sleep(2)

productionNum = {}
for i in range(1, len(productionRules)):
    productionNum[str(productionRules[i])] = i

 #---------------- Adding Closure ----------------

appendingClosure = findClosure("X->.S")
itemSet.append(appendingClosure)

#---------------- Implementing DFA ----------------

stateNumbers = {}
dfaRules = {}
numberofItems = 0

while True:
    if len(itemSet) == 0:
        break

    jk = itemSet.pop(0)
    kl = jk
    flag.append(jk)
    stateNumbers[str(jk)] = numberofItems
    numberofItems += 1

    if len(jk) > 1:
        for item in jk:
            jl = gotoFucntion(item)
            if jl not in itemSet and jl != kl:
                itemSet.append(jl)
                dfaRules[str(stateNumbers[str(jk)]) + " " + str(item)] = j
            else:
                dfaRules[str(stateNumbers[str(jk)]) + " " + str(item)] = j
```

```
for item in flag:
    for j in range(len(item)):
        if gotoFucntion(item[j]) not in flag:
            if item[j].index(".") != len(item[j]) - 1:
                flag.append(gotoFucntion(item[j]))

print("--------------------------------------------------------------------")
print("Total States: ", len(flag))
for i in range(len(flag)):
    print(i, ":", flag[i])
print("--------------------------------------------------------------------")
time.sleep(2)
```

```
dfa = {}
for i in range(len(flag)):
    if i in dfa:
        pass
    else:
        lst = getList(dfaRules, i)
        samp = {}
        for j in lst:
            s = j.split()[1].split('->')[1]
            search = s[s.index('.') + 1]
            samp[search] = stateNumbers[str(dfaRules[j])]

        if samp != {}:
            dfa[i] = samp

print(dfa)
time.sleep(2)
```

```
#---------------- Implementing Parsing Table ----------------
parsingTable = []
term = sorted(list(Terminals(productionRules)))
header = [''] * (len(term) + 1)
header[(len(term) + 1) // 2] = 'Action'

non_term = sorted(list(nonTerminals(productionRules)))
header2 = [''] * len(non_term)
header2[(len(non_term)) // 2] = 'Goto'

parsingTable.append([''] + term + non_term)

parsingTableDict = {}



for i in range(len(flag)):
    data = [''] * (len(term) + len(non_term))
    samp = {}

    #Action
    try:
        for j in dfa[i]:
            if not j.isupper() and j != '' and j != '.':
                ind = term.index(j)
                data[ind] = 'S' + str(dfa[i][j])
                samp[term[ind]] = 'S' + str(dfa[i][j])
```

```
except Exception:
    if i != 1:
        s = list(flag[i][0])
        s.remove('.')
        s = "".join(s)
        lst = [i] + ['r' + str(productionNum[s])] * len(term)
        lst += [''] * len(non_term)
        parsingTable.append(lst)
        for j in term:
            samp[j] = 'r' + str(productionNum[s])
    else:
        lst = [i] + [''] * (len(term) + len(non_term))
        lst[-1] = 'Accept'
        parsingTable.append(lst)

try:
    for j in dfa[i]:
        if j.isupper():
            ind = non_term.index(j)
            data[len(term) + ind] = dfa[i][j]

            samp[j] = str(dfa[i][j])

    parsingTable.append([i] + data)
except Exception:
    pass
```

```
  if samp == {}:
      parsingTableDict[i] = {'$': 'Accept'}
  else:
      parsingTableDict[i] = samp


final_table = tt.to_string(data=parsingTable, header=header + header2, style=tt.styles.ascii_thin_double, padding=(0, 1))

time.sleep(2)
print("\n")
print(final_table)
print("\n")

#---------------- String Parsing ----------------

string = input("Enter the string to be parsed: ")
string += '$'
print("\n")

stack = [0]
pointer = 0

header  = ['Process', 'Look Ahead', 'Symbol', 'Stack']
data = []
```

```
i = 0
accepted = False
while True:
    try:
        try:
            productions = dfa[stack[-1]]
            productionsNumber = productions[string[i]]
        except Exception:
            productionsNumber = None

        try:
            tab = parsingTableDict[stack[-1]]
            tabCheck = tab[string[i]]  # S or r
        except Exception:
            tab = parsingTableDict[stack[-2]]
            tabCheck = tab[stack[-1]]  # S or r

        if tabCheck == 'Accept':
            data.append(['Action({0}, {1}) = {2}'.format(stack[-1], string[i], tabCheck), i, string[i], str(stack)])
            accepted = True
            break
        else:
            if tabCheck[0] == 'S' and not str(stack[-1]).isupper():
                lst = ['Action({0}, {1}) = {2}'.format(stack[-1], string[i], tabCheck), i, string[i]]
                stack.append(string[i])
                stack.append(productionsNumber)
                lst.append(str(stack))
                data.append(lst)
                i += 1
            elif tabCheck[0] == 'r':
                lst = ['Action({0}, {1}) = {2}'.format(stack[-1], string[i], tabCheck), i, string[i]]
                x = None
                for i1 in productionNum:
                    if productionNum[i1] == int(tabCheck[1]):
                        x = i1
                        break

                length = 2 * (len(x.split('->')[1]))
                for _ in range(length):
                    stack.pop()

                stack.append(x[0])
                lst.append(str(stack))
                data.append(lst)

            else:
                lst = ['goto({0}, {1}) = {2}'.format(stack[-2], stack[-1], tabCheck), i, string[i]]
                stack.append(int(tabCheck))
                lst.append(str(stack))
                data.append(lst)
```

```
    except Exception:
        accepted = False
        break


try:
    parsing_table = tt.to_string(data=data, header=header, style=tt.styles.ascii_thin_double, padding=(0, 1))

    if accepted:
        string = string[:-1]

        print(parsing_table)
        print("The string {0} is parsable!".format(string))

    else:
        print("The string {0} is not parsable!".format(string))

except Exception:
    print("Invalid string entered!")

#---------------- End Of Program ----------------
```

OUTPUT:

```
Augmented Grammar
['X->.S', 'S->AA', 'A->aA', 'A->b']
--------------------------------------------------------------------
Total States:  7
0 : ['X->.S', 'S->.AA', 'A->.aA', 'A->.b']
1 : ['X->S.']
2 : ['S->A.A', 'A->.aA', 'A->.b']
3 : ['A->a.A', 'A->.aA', 'A->.b']
4 : ['A->b.']
5 : ['S->AA.']
6 : ['A->aA.']
--------------------------------------------------------------------
{0: {'S': 1, 'A': 2, 'a': 3, 'b': 4}, 2: {'A': 5, 'a': 3, 'b': 4}, 3: {'A': 6, 'a': 3, 'b': 4}}
```

	Action			Goto	
	$	a	b	A	S
0		S3	S4	2	1
1					Accept
2		S3	S4	5	
3		S3	S4	6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

```
Enter the string to be parsed: abb
```

Process	Look Ahead	Symbol	Stack
Action(0, a) = S3	0	a	[0, 'a', 3]
Action(3, b) = S4	1	b	[0, 'a', 3, 'b', 4]
Action(4, b) = r3	2	b	[0, 'a', 3, 'A']
goto(3, A) = 6	2	b	[0, 'a', 3, 'A', 6]
Action(6, b) = r2	2	b	[0, 'A']
goto(0, A) = 2	2	b	[0, 'A', 2]
Action(2, b) = S4	2	b	[0, 'A', 2, 'b', 4]
Action(4, $) = r3	3	$	[0, 'A', 2, 'A']
goto(2, A) = 5	3	$	[0, 'A', 2, 'A', 5]
Action(5, $) = r1	3	$	[0, 'S']
goto(0, S) = 1	3	$	[0, 'S', 1]
Action(1, $) = Accept	3	$	[0, 'S', 1]

The string abb is parsable!