Compiler Design

Exp–9 Computation of LR(0) Parser

Name:- K. DUSHYANT REDDY Reg No.:- RA1911033010029

Branch:- CSE-SE

Date:- 22/03/22

AIM: To implement Computation of LR(0) Parser.

CODE:

#LR(0) Parser

*def* closure(*I*,*nonT*):

    J = *I*

    for item in J :

        #print(item)

        index = item[1].index('.')

        if(index<(len(item[1])-1) and item[1][index+1] in *nonT*):

            for production in *nonT*[item[1][index+1]]:

                if( [item[1][index+1],str('.')+str(production)] not in J):

                    J.append([item[1][index+1],str('.')+str(production)])

    return J

state = []

I = []

*def* setOfItems(*start*,*nonTer*,*ter*):

    I.append(closure([['start','.'+*start*+'$']],*nonTer*))

*ter* += list(*nonTer*.keys())

    for conI in I:

        for grammar in *ter*:

            if(grammar == '$'):

                continue

            goto = False

            goto1 = False

            shift = False

            shift1 = False

            reduce = False

            close = []

            for item in conI:

                if(item[1].index('.')<(len(item[1])-1) and item[1][item[1].index('.')+1] == grammar):

                    close.append([item[0],item[1][:item[1].index('.')]+grammar+'.'+item[1][item[1].index('.')+2:]])

            l = closure(close,*nonTer*)

            if(len(l) == 0):

                continue

            if(grammar in *nonTer*.keys()):

                goto1 = True

            else:

                shift1 = True

            if(l not in I):

                if(goto1):

                    state.append(['g',I.index(conI)+1,len(I)+1,grammar])

                    goto = True

                elif(shift1):

                    shift = True

                    state.append(['s',I.index(conI)+1,len(I)+1,grammar])

                I.append(l)

            else:

                if(goto1):

                    goto = True

                    state.append(['g',I.index(conI)+1,I.index(l)+1,grammar])

                elif(shift1):

                    shift = True

                    state.append(['s',I.index(conI)+1,I.index(l)+1,grammar])

reduce = []

accept = -1

*def* toReduce(*rule*,*accept*,*start*):

    s = ['start',*start*+'.$']

    for parState in I:

        if(s in parState):

*accept* = I.index(parState)

        for item in parState:

            if( item in *rule*):

                reduce[I.index(parState)].append(*rule*.index(item))

    return *accept*

symbolMap = dict()

parseTable = []

*def* createParseTable(*ter*):

    for i in state:

        parseTable[i[1]-1][symbolMap[i[3]]] = i[0]+str(i[2]-1)

    parseTable[accept][symbolMap['$']] = 'a'

    for i in reduce:

        if(len(i)>0):

            for j in *ter*:

                parseTable[reduce.index(i)][symbolMap[j]] = 'r'+str(i[0])

*class* Stack:

*def* \_\_init\_\_(*self*):

*self*.\_\_storage = []

*def* isEmpty(*self*):

        return len(*self*.\_\_storage) == 0

*def* push(*self*,*p*):

*self*.\_\_storage.append(*p*)

*def* pop(*self*):

        return *self*.\_\_storage.pop()

*def* top(*self*):

        return *self*.\_\_storage[len(*self*.\_\_storage) - 1]

*def* length(*self*):

        return len(*self*.\_\_storage)

*def* \_\_str\_\_(*self*):

        """

        Because of using list as parent class for stack, our last element will

        be first for stack, according to FIFO principle. So, if we will use

        parent's implementation of str(), we will get reversed order of

        elements.

        """

        return 'stack [{}]'.format(', '.join([ str(i) for i in reversed(*self*.\_\_storage) ]))

*def* parseString(*rule*,*string*):

    index = 0

    flag = False

    st = Stack()

    st.push('0')

    while(index < len(*string*)):

        print(st , *string* , index , *sep* = '\t\t ')

        c = parseTable[int(st.top())][symbolMap[*string*[index]]][0]

        if(c == 'a'):

            flag = True

            break

        pt = parseTable[int(st.top())][symbolMap[*string*[index]]][1:]

        pt = int(int(pt))

        if( c == 'r'):

            l = len(*rule*[pt][1])

            l \*= 2

            l -= 2

            if(l >= st.length()):

                break

            else:

                for i in range(l):

                    st.pop()

                top = int(st.top())

                st.push(*rule*[pt][0])

                st.push(parseTable[top][symbolMap[st.top()]][1:])

        else:

            st.push(*string*[index])

            st.push(str(pt))

            index+=1

    return flag

terminals = []

nonTerminals = dict()

terminals = input("Enter Terminals (|) : ").split("|")

n = int(input("No. of Non - Terminals  : "))

for i in range(n):

    ch = input("NonTerminals : ").strip()

    rules = input("Productions (|) : ").split("|")

    nonTerminals[ch] = rules

S = input("Start Symbol :  ")

terminals+=['$']

print("Productions : ")

for i in nonTerminals.keys():

    print(i,"-->",*end*=' ')

    for j in nonTerminals[i]:

        print(j,*end*= ' | ')

    print()

setOfItems(S,nonTerminals,terminals)

print("canonicals Items : ")

for count , i in enumerate(I):

    print(count+1 , i)

print("state Transitions : ")

for count , i in enumerate(state):

    print(count+1, i)

rule = []

accept = -1

for i in nonTerminals.keys():

    for j in nonTerminals[i]:

        rule.append([i,j+str('.')])

print('rule :')

for i in rule:

    print(i)

reduce = [ [] for i in range(len(I)) ]

accept = toReduce(rule,accept,S)

print("reduce")

for count,i in enumerate(reduce):

    print(count+1,i)

print("accept : ",accept+1)

symbols = []

symbols += terminals

for count , i in enumerate(symbols):

    symbolMap[i] = count

print(symbols)

parseTable = [ ['-' for i in range(len(symbols))] for j in range(len(I)) ]

for i in nonTerminals.keys():

    terminals.remove(i)

createParseTable(terminals)

print('Parse Table')

print(" \t\t",*end*='')

for i in symbols:

    print(i,*end*= '\t')

print()

for count,j in enumerate(parseTable):

    print(count,*end*='\t\t')

    for i in j:

        print(i,*end*='\t')

    print()

string = input("String : ")

string+='$'

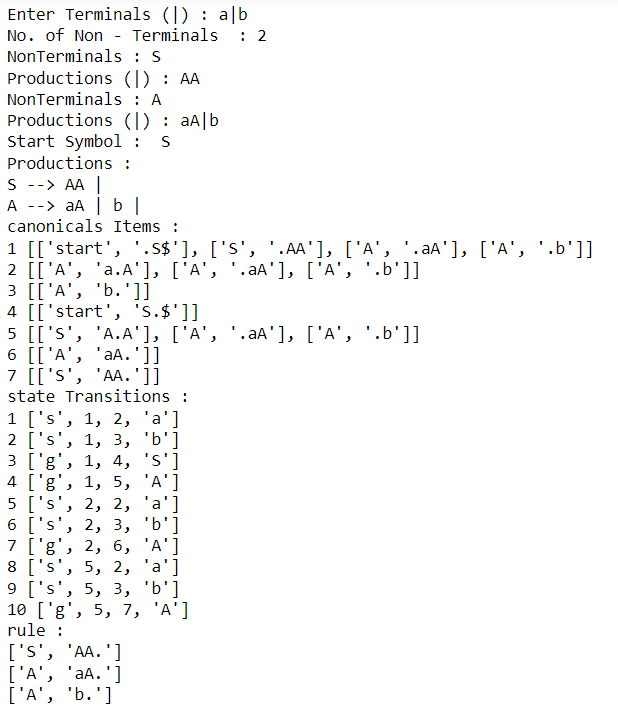
if(parseString(rule,string)):

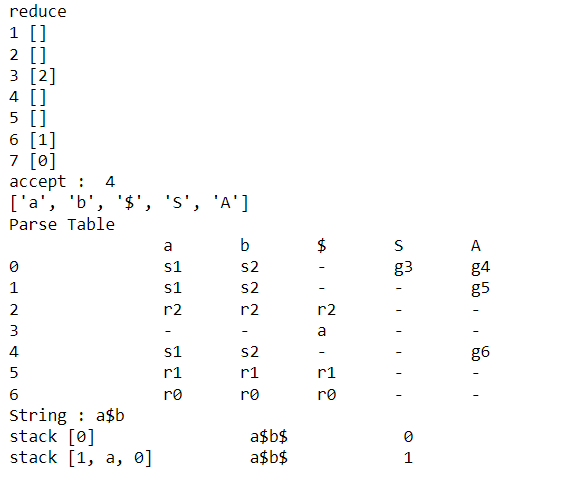
    print("accepted")

else:

    print("Not accepted")

OUTPUT:





RESULT:

The code is successfully implemented in Python language.