Department of Computer Science and Engineering, SVNIT Surat
System Software
Lab Assignment -7

U20CS005
BANSI MARAKANA

1. Write a program to construct LR (1) parse table for the following grammar and check whether the given input can be accepted or not.
Grammar:
S -> AaAb | BbBa
A -> ε
B -> ε

```
from collections import OrderedDict
tl=OrderedDict()
ntl=OrderedDict()
productionRules=[]
ntList, tList=[], []

class Terminal:
    def __init__(self, symbol):
        self.symbol=symbol
    def __str__(self):
        return self.symbol

class NonTerminal:
    def __init__(self, symbol):
        self.symbol=symbol
        self.first=set()
        self.follow=set()
    def __str__(self):
        return self.symbol

# |= is Unioning
    def add_first(self, symbols):
        self.first |= set(symbols)
    def add_follow(self, symbols):
        self.follow |= set(symbols)

def computeFirst(symbol): # Function accepts a symbol, returns its First set
    global productionRules, ntl, tl
    #1. Symbol is a Terminal
    if symbol in tl:
        return set(symbol)
    for prod in productionRules:
```

```
        head, body=prod.split('->')
        if head!=symbol:
            continue
    #2. RHS is an epsilon
        if body=='':
            ntl[symbol].add_first('ϵ')
            continue
    #3 X-> Y1 Y2 Y3....Yn
        for i, Y in enumerate(body):
            if body[i]==symbol:  # If it is the same symbol, skip to next (Left recursion)
                continue
            t=computeFirst(Y)
            # Adding the first of this Non Terminal
            ntl[symbol].add_first(t-set('ϵ'))
            if 'ϵ' not in t:    # if epsilon not in the first of this NT, stop
                break
            if i==len(body)-1:        # Reached the end, adding epsilon
                ntl[symbol].add_first('ϵ')
    return ntl[symbol].first

def getFirst(symbol): #getter
    return computeFirst(symbol)

def computeFollow(symbol):
    global productionRules, ntl, tl
    #1 If the symbol is start symbol, appending $
    if symbol == list(ntl.keys())[0]: # Checking if it is the start
        ntl[symbol].add_follow('$')
    for prod in productionRules:
        head, body=prod.split('->')
        for i, B in enumerate(body):
            if B != symbol:
                continue
    # Locating the symbol
    # Say A -> aBB'B1', follow(B) = non-epsilon symbols in first(B')
            if i != len(body)-1:
                ntl[symbol].add_follow(getFirst(body[i+1]) - set('ϵ'))
    # if A -> aBb where first(b) contains epsilon, or A -> aB then follow(B) = follow (A)
            if i == len(body)-1 or 'ϵ' in getFirst(body[i+1]) and B != head:
                ntl[symbol].add_follow(getFollow(head))

def getFollow(symbol):  #getter
    global ntl, tl
    if symbol in tl.keys():
```

```
        return None
    return ntl[symbol].follow


def getProductions():
    pl=None
    print("'Enter the grammar production (Enter 'end' to stop input)
#(Enter in format: "A->Y1Y2..Yn" OR "A->" {for epsilon})"')
    global productionRules, tl, ntl
    ctr=1
    if pl==None:
        while True:
            productionRules.append(input().replace(' ', ''))
            if productionRules[-1].lower() in ['end', '']:
                del productionRules[-1]
                break
            head, body=productionRules[ctr-1].split('->')
            if head not in ntl.keys():
                ntl[head]=NonTerminal(head)
            #for all terminals in the body of the production
            for i in body:
                if not 65<=ord(i)<=90:
                    if i not in tl.keys():
                        tl[i]=Terminal(i)
            #for all non-terminals in the body of the production
                elif  i not in ntl.keys():
                    ntl[i]=NonTerminal(i)
            ctr+=1
    return pl

class State:
# This class is for states/items.
    _id=0
    def __init__(self, closure):
        self.closure=closure
        self.no=State._id
        State._id+=1

class Item(str):
# This class is for every single production rule in a single state/item.
    def __new__(cls, item, lookAhead=list()):
        self=str.__new__(cls, item)
        self.lookAhead=lookAhead
        return self
```

```
    def __str__(self):
        return super(Item, self).__str__()+", "+'|'.join(self.lookAhead)

def closure(items):
# This function finds closure of given production rules
    def exists(newitem, items):
    #This function is to check whether a production rule is already available in a given item
        for i in items:
            if i==newitem and sorted(set(i.lookAhead))==sorted(set(newitem.lookAhead)):
                return True
        return False
    global productionRules
    while True:
    # This loop finds closure of given production rule
        flag=0
        for i in items:
        # This loop iterates till all production rules are exhausted in a given item
            if i.index('.')==len(i)-1:
                continue # This checks whether dot is at the end so that we can skip that rule
            Y=i.split('->')[1].split('.')[1][0]
            # We store whatever is available after dot in Y.
            if i.index('.')+1<len(i)-1:
            # a.Ab,$ ,it makes first(b) as lookahead of the production rules that emerges from this
rule
                lastr=list(computeFirst(i[i.index('.')+2])-set(chr(1013)))
            else:
            #If production rule is of type A-> a.A, as lookahead of the production rules that emerges
from this rule
                lastr=i.lookAhead
            for prod in productionRules:
            #If production rule is of type A-> a.Bc, it checks for production rule that has B on LHS
                head, body=prod.split('->')
                if head!=Y:
                    continue #skips rule that don't have B on LHS
                newitem=Item(Y+'->.'+body, lastr)
                # Add the production rule with '.' at beginning
                if not exists(newitem, items):
                #Check if the rule is already available, if not add it to the state
                    items.append(newitem)
                    flag=1
        if flag==0: break
    return items

def goTo(items, symbol):
```

```
#This function finds goto of items on symbol
    global productionRules
    newState=[]
    #the new state is stored in newState
    for i in items:
        if i.index('.')==len(i)-1:
            continue
        #If "." is at end , it can be ignored
        head, body=i.split('->')
        seen, unseen=body.split('.')
        if unseen[0]==symbol and len(unseen) >= 1:
        #Shift dot to one place
            newState.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:], i.lookAhead))
    return closure(newState)

def computeStates():
# This function gives the automaton
    def contains(states, t):
    # This function checks whether a new state formed is already discovered
        for s in states:
            if len(s) != len(t): continue
            if sorted(s)==sorted(t):
                for i in range(len(s)):
                        if s[i].lookAhead!=t[i].lookAhead:
                            break
                else: return True
        return False
    global productionRules, ntList, tList
    head, body= productionRules[0].split('->')
    # For the start production rule, lookahead is '$'
    states=[closure([Item(head+'->.'+body, ['$'])])]
    while True:
    # This iterates till no new states are formed
        flag=0
        for s in states:
            for e in ntList+tList:
                t=goTo(s, e)
                # Finds goto of s on e
                if t == [] or contains(states, t):
                    continue
                # Checks if t is already present or empty ,if not we can append t to states
                states.append(t)
                flag=1
        if not flag: break
```

```
    return states

def makeTable(states):
#To create CLR Table
    global ntList, tList
    def getStateNo(t):
    # To get state number of a given state
        for s in states:
            if len(s.closure) != len(t): continue
            if sorted(s.closure)==sorted(t):
                for i in range(len(s.closure)):
                        if s.closure[i].lookAhead!=t[i].lookAhead: break
                else: return s.no
        return -1

    def getProdNo(closure):
    # To get production number
        closure=''.join(closure).replace('.', '')
        return productionRules.index(closure)
    clrTable=OrderedDict()

    for i in range(len(states)):
    # To give state number
        states[i]=State(states[i])
    for s in states:
    # Creates rows
        clrTable[s.no]=OrderedDict()
        for item in s.closure:
            head, body=item.split('->')
            if body=='.':
            #To handle production of type S->.,$
                for term in item.lookAhead:
                    if term not in clrTable[s.no].keys():
                        clrTable[s.no][term]={'r'+str(getProdNo(item))}
                    else: clrTable[s.no][term] |= {'r'+str(getProdNo(item))}
                continue
            nextSym=body.split('.')[1]
            if nextSym==='':
            # To handle production of type S->ab.,a
                if getProdNo(item)==0:
                # Checks if it is accepting state
                    clrTable[s.no]['$']='AC'
                else:
                    for term in item.lookAhead:
```

```
                    if term not in clrTable[s.no].keys():
                        clrTable[s.no][term]={'r'+str(getProdNo(item))}
                    else: clrTable[s.no][term] |= {'r'+str(getProdNo(item))}
                continue
            nextSym=nextSym[0]
            t=goTo(s.closure, nextSym)
            if t != []:
            #To handle production of type A-> ab.c,a|b
                if nextSym in tList:
                    if nextSym not in clrTable[s.no].keys():
                        clrTable[s.no][nextSym]={'s'+str(getStateNo(t))}
                    else: clrTable[s.no][nextSym] |= {'s'+str(getStateNo(t))}
                else: clrTable[s.no][nextSym] = str(getStateNo(t))
    return clrTable

def augmentGrammar():
                    #Adding the extra production rule
    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in ntList:
            startProd=productionRules[0]
            productionRules.insert(0, chr(i)+'->'+startProd.split('->')[0])
            return

def main():
    global productionRules, ntl, ntList, tl, tList
    # We start with getting the grammar from the user
    getProductions()

    print("\t**********FIRST And FOLLOW Of Non-Terminals:**********")
    # We then compute first and follow of the non-terminals
    for nt in ntl:
        computeFirst(nt)
        computeFollow(nt)
        print(nt)
        print("\tFirst:\t", getFirst(nt))
        print("\tFollow:\t", getFollow(nt), "\n")

    #We augment the grammar with an extra production
    augmentGrammar()
    ntList=list(ntl.keys())
    tList=list(tl.keys()) + ['$']
    print('Non-terminals: ',end=' ')
    print(*ntList,sep=', ')       #Printing the non-terminals
    print('Terminals: ',end=' ')
```

```
print(*tList,sep=', ')        #Printing the terminals

#The automaton is created by computing the states
automaton=computeStates()
ctr=0
for s in automaton:
    print("State{}:".format(ctr))
    for i in s:
        print("\t", i)
    ctr+=1

table=makeTable(automaton)    #creates the CLR(1) Table
print('_____________________________________________________________________________________')
print("\n**********\tCLR(1) Parsing Table**********\n")
symList = ntList + tList     #list of all the symbols
shiftReduce, reduceReduce=0, 0
print('_____________________________________________________________________________________')
print('\t|  ','\t|  '.join(symList),'\t\t|')
print('_____________________________________________________________________________________')

for i, j in table.items():
    temp = []
    for sym in symList:
        if type(j.get(sym)) in (str,None):
            temp.append(j.get(sym,' '))
        else:                                #type is a 'set'
            temp.append(next(iter(j.get(sym,' '))))
    print("I"+str(i), "\t|  ",'\t|  '.join(temp),'\t\t|')
    shifts, reduces=0, 0
    for p in j.values():
        if p!='AC' and len(p)>1:
            p=list(p)
            if('r' in p[0]):
                reduces+=1
            else: shifts+=1
            if('r' in p[1]):
                reduces+=1
            else: shifts+=1
    if reduces>0 and shifts>0:
        shiftReduce+=1
    elif reduces>0:
        reduceReduce+=1
print('_____________________________________________________________________________________')
print("\n", shiftReduce, "s/r conflicts |", reduceReduce, "r/r conflicts")
```

```
    if(shiftReduce>0 or reduceReduce>0):
        print("\n\nNot a CLR(1) grammar\n")
    else:                                   #get inputs and parse only if it is a CLR(1) grammar

print('_____________________________________________________________________________')
        ch='c'
        while(ch!='q'):
            print("Enter the string to be parsed: ")        #get input for parsing
            Input=input()+'$'
            try:
                stack=['0']                         #initialize stack
                symbol=[]
                a=list(table.items())
                print("Productions\t:",productionRules)
                print('Stack',"\t\t\t\t",'Symbol',"\t\t\t\t",'Input',"\t\t\t\t",'Rule')
                print(*stack,"\t\t\t\t",*symbol,"\t\t\t\t",*Input,sep="")
                while(len(Input)!=0):
                    b=list(a[int(stack[-1])][1][Input[0]])
                    if(b[0][0]=="s" ):                  #If Shift operation
                        symbol.append(Input[0])
                        stack.append(b[0][1:])              #Push the value next to S e.g. S2, 2 is pushed
                        Input=Input[1:]
                        print(*stack,"\t\t\t\t",*symbol,"\t\t\t\t",*Input,sep="")
                    elif(b[0][0]=="r" ):
                        s=int(b[0][1:])
                        l=len(productionRules[s])-3         #length of RHS of production rule
                        prod=productionRules[s]
                        l1=len(symbol)-l
                        l=len(stack)-l
                        symbol=symbol[:l1]
                        stack=stack[:l]
                        s=a[int(stack[-1])][1][prod[0]]
                        symbol.append(prod[0])
                        stack.append(s)
                        print(*stack,"\t\t\t\t",*symbol,"\t\t\t\t",*Input,"\t\t\t\t",prod,sep="")
                    elif(b[0][0]=="A"):
                        print("\n\tString Accepted\n")
                        break
            except:
                print('\n\tString is not accepted by Parser!!\n')
            print("Press c to continue or q to quit")
            ch=input()
if __name__=="__main__":
    main()
```

```
Enter the grammar production (Enter 'end' to stop input)
#(Enter in format: "A->Y1Y2..Yn" OR "A->" {for epsilon})
S->AaAb
S->BbBa
A->
B->
end
          ***********FIRST And FOLLOW Of Non-Terminals:***********
S
          First:   {'a', 'b'}
          Follow:  {'$'}

A
          First:   {'ϵ'}
          Follow:  {'a', 'b'}

B
          First:   {'ϵ'}
          Follow:  {'a', 'b'}

Non-terminals:  S, A, B
Terminals:  a, b, $
```

```
State0:
          Z->.S, $
          S->.AaAb, $
          S->.BbBa, $
          A->., a
          B->., b
State1:
          Z->S., $
State2:
          S->A.aAb, $
State3:
          S->B.bBa, $
State4:
          S->Aa.Ab, $
          A->., b
State5:
          S->Bb.Ba, $
          B->., a
State6:
          S->AaA.b, $
State7:
          S->BbB.a, $
State8:
          S->AaAb., $
State9:
          S->BbBa., $
```

**********CLR(1) Parsing Table**********

	S	A	B	a	b	$
I0	1	2	3	r3	r4	
I1						AC
I2				s4		
I3					s5	
I4		6			r3	
I5			7	r4		
I6					s8	
I7				s9		
I8						r1
I9						r2

---

0 s/r conflicts | 0 r/r conflicts

```
Enter the string to be parsed:
ba
Productions     : ['Z->S', 'S->AaAb', 'S->BbBa', 'A->', 'B->']
Stack                           Symbol                          Input                           Rule
0                                                               ba$
03                              B                               ba$                             B->
035                             Bb                              a$
0357                            BbB                             a$                              B->
03579                           BbBa                            $
01                              S                               $                               S->BbBa

        String Accepted
```