Compiler Design

Exp–7 Shift Reduce Parsing

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

Branch:- CSE-SE

Date:- 08/03/22

AIM: To implement Shift Reduce Parsing

CODE:

#Shift Reduce Parsing

from follow import follow\_pos

from state import \*

from numpy import nan

from pandas import DataFrame as df, MultiIndex

states = []

parsing\_table = df()

*def* grammar\_fromstr(*g*):

    grm = *g*.split('\n')

    rules = []

    for r in grm:

        if r.strip()=='':continue

        lhs, rhs\_ls = r.split('=>')

        for rhs in rhs\_ls.split('|'):

            rules.append((lhs.strip(),rhs.strip().split()))

    return rules

*def* augment(*grammar\_str*):

    '''

    - Rules must be in form LHS => RHS

    - NONTERMINALS must start with (`) and be UPPERCASE. use only [A-Z] and (\_) for NONTERMINALS

    - Terminals must be lower case

    - If there are many rhs for one lhs, you should put them all in the same line and separate them with (|) i.e: X => Y | Z | !εpsilon

    - Use the form (!εpsilon) for writing epsilons

    '''

    grammar= grammar\_fromstr(*grammar\_str*)

    rhs = [grammar[0][0]]

    aug=Rule(grammar[0][0]+"'", tuple(rhs))

    s = State()

    s.add\_rule(aug)

    Rule.augmented.append(aug)

    for r in grammar:

        Rule.augmented.append(Rule(r[0],r[1]))

    return s, extract\_symbols(grammar)

*def* extract\_symbols(*rules*):

    terminals = []

    non\_terminals = []

    for r in *rules*:

        if r[0] not in non\_terminals:

            non\_terminals.append(r[0])

        for s in r[1]:

            if not s.startswith('`'):

                if s not in terminals:

                    if s != '!εpsilon':

                        terminals.append(s)

            else:

                if s not in non\_terminals:

                    non\_terminals.append(s)

    terminals.append('$')

    return (non\_terminals, terminals)

*def* goto\_operation():

    for s in states:

        transitions = []

        for r in  s.rules:

            rule = r.movedot()

            dotatend = rule == None

            if dotatend:

                continue

            transitions.append((r.handle(), rule))

        gotoself(transitions, s)

        for t in transitions:

            items\_same\_X = [r for r in transitions if r[0] == t[0]]

            make\_transition(s, items\_same\_X)

    return State.graph

*def* gotoself(*transitions*, *s*):

    for t in *transitions*:

        if t[1] in *s*.rules:

*s*.goto( *s*.\_i, t[0])

*transitions*.remove(t)

*def* make\_transition(*source*, *items\_same\_X*):

    new\_state = newstate(*items\_same\_X*)

    exists = False

    for s in states:

        if new\_state == s:

*source*.goto(s.\_i, *symbol*=*items\_same\_X*[0][0])

            exists=True

            State.\_count = State.\_count-1

            break

    ###

    if not exists:

        new\_state.closure()

        states.append(new\_state)

*source*.goto(new\_state.\_i, *symbol*=*items\_same\_X*[0][0])

*def* newstate(*items\_same\_X*):

    new\_state = State()

    for r in *items\_same\_X*:

        new\_state.add\_rule(r[1])

    return new\_state

*def* parsing\_table\_skelton(*non\_terminals*, *terminals*):

    levels = (['action']\*len(*terminals*) + ['goto']\*len(*non\_terminals*))

    columns = *terminals*+*non\_terminals*

    index = [s.\_i for s in states]

    return df(*index*=index, *columns*=MultiIndex.from\_tuples(list(zip(levels,columns)),*names*=['table','symbol'])).fillna('\_')

*def* slr\_parsing\_table(*items*):

    global parsing\_table

    for i in *items*:

        isterminal = not i[2].startswith('`')

        if isterminal: # Shift

            cell = parsing\_table.loc[(i[0]), ('action', i[2])]

            if cell !='\_':

                print('conflict: '+ cell + '    s'+str(i[1]))

                continue

            parsing\_table.loc[(i[0]), ('action', i[2])] = 's'+str(i[1])

        else:

            parsing\_table.loc[(i[0]), ('goto', i[2])] = i[1]

    n = Rule.\_n

    reduce = [(s.rules[0].lhs, s.\_i, Rule.augmented.index(s.rules[0].copy())) for s in states if s.hasreduce]

    for r in reduce:

        if r[0].endswith("'"):

            parsing\_table.loc[(r[1]), ('action', '$')] = 'accept'

        else:

            for f in follow\_pos(r[0]):

                cell = parsing\_table.loc[(r[1]), ('action', f)]

                if cell !='\_':

                    print('conflict: '+cell + '    r'+str(r[2]+n))

                parsing\_table.loc[(r[1]), ('action', f)] = 'r'+str(r[2]+n)

*def* moves(*s*):

    snap=[]

    stack = [('$',State.\_n)]

    input\_ = *s*.split()+['$']

    action = []

    while True:

        a = parsing\_table.loc[(stack[-1][1]), ('action', input\_[0])]

        action.append(a)

        snap.append((''.join([s[0]+str(s[1]) for s in stack]), ' '.join(input\_)))

        if a == 'accept':

            print ('Driver: accept')

            break

        #Shift

        if a.startswith('s'):

            stack.append((input\_[0],int(''.join(a[1:]))))

            input\_.remove(input\_[0])

        #Reduce

        elif a.startswith('r'):

            r = Rule.augmented[(int(''.join(a[1:])))]

            for \_ in range(len(r.rhs)):

                stack.pop()

            goto = parsing\_table.loc[(stack[-1][1]), ('goto', r.lhs)]

            stack.append((r.lhs, goto))

            action[-1] = ' '.join([a,'goto:'+str(goto), str(r).replace(' • ', ' ')])

        else:

            print('Driver: Syntax error')

            break

    return df(*data*=list(zip([s[0] for s in snap],[s[1]for s in snap],action)) ,*columns*=('Stack','Input','Action'))

*def* run(*grammar*):

    global parsing\_table

    start\_state, symbols = augment(*grammar*)

    start\_state.closure()

    states.append(start\_state)

    items = goto\_operation()

    parsing\_table =  parsing\_table\_skelton(symbols[0], symbols[1])

    slr\_parsing\_table(items)

    return items

*def* test(*grammar*, *test\_string*):

    states\_graph = run(*grammar*)

    for s in states:

        print(s, *end*='\n')

    print(parsing\_table)

    driver\_table = moves(*test\_string*)

    print(driver\_table)

if \_\_name\_\_ == '\_\_main\_\_':

    print(augment.\_\_doc\_\_)

    g4 = """`E => `E + `T

    `E => `T

    `T => `T \* `F

    `T => `F

    `F => ( `E )

    `F => id"""

    print(g4, *end*='\n------grammar------\n\n')

    test(g4, 'id + id \* id')

Subfiles:

follow.ipynb

from state import Rule

\_first = {}

\_follow = {}

*def* first\_pos(*symbol*):

    first = set()

    if not *symbol*.startswith('`'):

        return set([*symbol*])

    for r in [i for i in Rule.augmented if i.lhs == *symbol*]:

        whole\_rhs\_has\_e = True

        for s in r.rhs:

            has\_e=False

            for f in first\_pos(s):

                if f == '!εpslon':

                    has\_e=True

                else:

                    first.add(f)

            if not has\_e:

                whole\_rhs\_has\_e=False

                break

        if whole\_rhs\_has\_e:

            first.add('!εpslon')

    global \_first

    \_first[*symbol*]=list(first)

    return list(first)

*def* follow\_pos(*symbol*, *A*=None):

    follow = set()

    if *symbol*.endswith("'"):

        follow.add('$')

    for r in [i for i in Rule.augmented if *symbol* in i.rhs]:

        occurences = r.rhs.count(*symbol*)

        if *symbol* == r.rhs[-1]:

            if *symbol* != r.lhs and *A* != r.lhs:

                for f in follow\_pos(r.lhs, *symbol*):

                    follow.add(f)

            continue

        beta = r.rhs

        for i in range(occurences):

            j=beta.index(*symbol*)

            beta = r.rhs[j+1:]

            s=beta[0]

            f = first\_pos(s)

            for f1 in f:

                if f1 == '!εpsilon':

                    if *symbol* == r.lhs or *A* == r.lhs: continue

                    for f2 in follow\_pos(r.lhs, *symbol*):

                        follow.add(f2)

                else:

                    follow.add(f1)

    global \_follow

    \_follow[*symbol*]=list(follow)

    return follow

*def* test\_frstfllw(*symbols*):

    c=[]

    for s in *symbols*:

        f = first\_pos(s)

        fo = follow\_pos(s)

        c.append([s,' '.join(f),' '.join(fo)])

    return c

state.ipynb

*class* State:

    \_n=0

    \_count=0+\_n

    graph=[]

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

*self*.rules = []

*self*.\_i = State.\_count

*self*.hasreduce=0

        State.\_count = State.\_count+1

*def* add\_rule(*self*, *rule*):

        if *rule* not in *self*.rules:

*self*.rules.append(*rule*)

            if *rule*.\_closure==-1:

*self*.hasreduce=1

*def* goto(*self*, *distination\_index*, *symbol*):

        g = [*self*.\_i, *distination\_index*, *symbol*]

        if g not in State.graph:

            State.graph.append(g)

*def* closure(*self*):

        for rule in *self*.rules:

            if rule.visited:

                continue

            for r in rule.visit():

                if r not in *self*.rules:

*self*.add\_rule(r)

*def* \_\_eq\_\_(*self*, *s*):

        "If self rules in s rules"

        if not isinstance(*s*, State):

            return NotImplemented

        eq = True

        if *self*.rules.\_\_len\_\_() > *s*.rules.\_\_len\_\_():

            return False

        for r in *self*.rules:

            eq = eq and (r in *s*.rules)

        return eq

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

        s = []

        max\_len=1

        for r in *self*.rules:

            line='    ['+str(r)

            s.append(line)

            if len(line) > max\_len: max\_len=len(line)

        for i in range(len(s)):

            pad = max\_len-len(s[i])

            s[i]=s[i]+' '\*pad+']'

        s.insert(0,''.join(['I',str(*self*.\_i),':',' '\*(max\_len-2)]))

        return '\n'.join(s)

*class* Rule:

    \_n=0

    augmented = []

*def* \_\_init\_\_(*self*, *lhs*, *rhs*=[], *dot\_index*=0):

*self*.lhs = *lhs*

        if *rhs* == ['!εpslon']:

*self*.rhs=[]

        else:

*self*.rhs = *rhs*

*self*.\_closure = *dot\_index*

        if *self*.dotatend():

*self*.\_closure = -1

*self*.visited = 0

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

        rhs = list(*self*.rhs)

        dot = *self*.\_closure

        if dot == -1:

            dot = len(rhs)

        rhs.insert(dot, '•')

        return *self*.lhs + ' → ' + ' '.join(rhs)

*def* \_\_eq\_\_(*self*, *rule*):

        if not isinstance(*rule*, Rule):

            return NotImplemented

        return *self*.lhs == *rule*.lhs and *self*.rhs == *rule*.rhs and *self*.\_closure == *rule*.\_closure

*def* handle(*self*):

        return *self*.rhs[*self*.\_closure]

*def* visit(*self*):

*self*.visited = 1

        if *self*.\_closure != -1:

            handle =  *self*.rhs[*self*.\_closure]

            if handle.startswith('`'):

                return [r.copy() for r in Rule.augmented if r.lhs == handle]

        return []

*def* dotatend(*self*):

        if *self*.\_closure == len(*self*.rhs):

            return True

        return False

*def* movedot(*self*):

        if *self*.\_closure == -1:

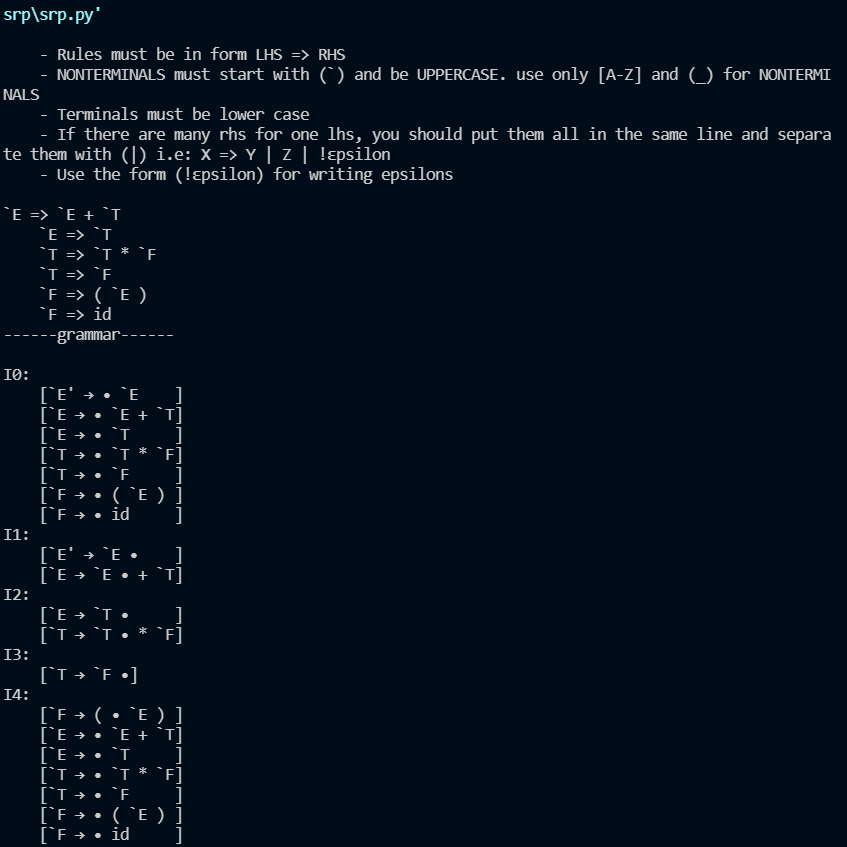
            return None

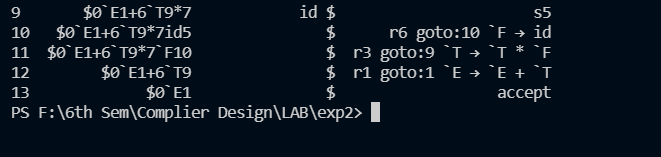
        return Rule(*self*.lhs, *self*.rhs, *self*.\_closure + 1)

*def* copy(*self*):

        return Rule(*self*.lhs,*self*.rhs)

OUTPUT:





RESULT:

The code is successfully implemented in Python language.