

COMPILER

MINI PROJECT

Aim: Design a predictive parser for a given language

Code:

```
def removeLeftRecursion(rulesDiction):
```

```
    store = { }
```

```
    for lhs in rulesDiction:
```

```
        alphaRules = []
```

```
        betaRules = []
```

```
        allrhs = rulesDiction[lhs]
```

```
        for subrhs in allrhs:
```

```
            if subrhs[0] == lhs:
```

```
                alphaRules.append(subrhs[1:])
```

```
            else:
```

```
                betaRules.append(subrhs)
```

```
        if len(alphaRules) != 0:
```

```
            lhs_ = lhs + ""
```

```
            while (lhs_ in rulesDiction.keys()) \
```

```
                or (lhs_ in store.keys()):
```

```
                lhs_ += ""
```

```
        for b in range(0, len(betaRules)):
```

```
            betaRules[b].append(lhs_)
```

```
rulesDiction[lhs] = betaRules
```

```
for a in range(0, len(alphaRules)):
```

```
    alphaRules[a].append(lhs_)
```

```
alphaRules.append(['#'])
```

```
store[lhs_] = alphaRules
```

```
for left in store:
```

```
    rulesDiction[left] = store[left]
```

```
return rulesDiction
```

```
def LeftFactoring(rulesDiction):
```

```
newDict = { }
```

```
for lhs in rulesDiction:
```

```
    allrhs = rulesDiction[lhs]
```

```
    temp = dict()
```

```
    for subrhs in allrhs:
```

```
        if subrhs[0] not in list(temp.keys()):
```

```
            temp[subrhs[0]] = [subrhs]
```

```
        else:
```

```
            temp[subrhs[0]].append(subrhs)
```

```
new_rule = []

tempo_dict = {}

for term_key in temp:

    allStartingWithTermKey = temp[term_key]
    if len(allStartingWithTermKey) > 1:

        lhs_ = lhs + ""
        while (lhs_ in rulesDiction.keys()) \
            or (lhs_ in tempo_dict.keys()):
            lhs_ += ""

        new_rule.append([term_key, lhs_])

    ex_rules = []
    for g in temp[term_key]:
        ex_rules.append(g[1:])
    tempo_dict[lhs_] = ex_rules
else:

    new_rule.append(allStartingWithTermKey[0])

newDict[lhs] = new_rule

for key in tempo_dict:
    newDict[key] = tempo_dict[key]
return newDict
```

```
def first(rule):  
    global rules, nonterm_userdef, \  
        term_userdef, diction, firsts  
  
    if len(rule) != 0 and (rule is not None):  
        if rule[0] in term_userdef:  
            return rule[0]  
        elif rule[0] == '#':  
            return '#'
```

```
    if len(rule) != 0:  
        if rule[0] in list(diction.keys()):  
            fres = []  
            rhs_rules = diction[rule[0]]  
            for itr in rhs_rules:  
                indivRes = first(itr)  
                if type(indivRes) is list:  
                    for i in indivRes:  
                        fres.append(i)  
                else:  
                    fres.append(indivRes)
```

```
    if '#' not in fres:  
        return fres  
    else:
```

```
newList = []
fres.remove('#')
if len(rule) > 1:
    ansNew = first(rule[1:])
    if ansNew != None:
        if type(ansNew) is list:
            newList = fres + ansNew
        else:
            newList = fres + [ansNew]
    else:
        newList = fres
return newList

fres.append('#')
return fres
```

```
def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    solset = set()
    if nt == start_symbol:

        solset.add('$')

    for curNT in diction:
```

```
rhs = diction[curNT]

for subrule in rhs:
    if nt in subrule:

        while nt in subrule:
            index_nt = subrule.index(nt)
            subrule = subrule[index_nt + 1:]

        if len(subrule) != 0:

            res = first(subrule)

            if '#' in res:
                newList = []
                res.remove('#')
                ansNew = follow(curNT)
                if ansNew != None:
                    if type(ansNew) is list:
                        newList = res + ansNew
                    else:
                        newList = res + [ansNew]
                else:
                    newList = res
            res = newList
        else:

            if nt != curNT:
                res = follow(curNT)
```

```
        if res is not None:
            if type(res) is list:
                for g in res:
                    solset.add(g)
            else:
                solset.add(res)
    return list(solset)
```

```
def computeAllFirsts():
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        # remove un-necessary spaces
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        # remove un-necessary spaces
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs

    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")
    print(f"\nAfter elimination of left recursion:\n")
```

```
diction = removeLeftRecursion(diction)
```

```
for y in diction:
```

```
    print(f"{y}->{diction[y]}")
```

```
print("\nAfter left factoring:\n")
```

```
diction = LeftFactoring(diction)
```

```
for y in diction:
```

```
    print(f"{y}->{diction[y]}")
```

```
for y in list(diction.keys()):
```

```
    t = set()
```

```
    for sub in diction.get(y):
```

```
        res = first(sub)
```

```
        if res != None:
```

```
            if type(res) is list:
```

```
                for u in res:
```

```
                    t.add(u)
```

```
            else:
```

```
                t.add(res)
```

```
    firsts[y] = t
```

```
print("\nCalculated firsts: ")
```

```
key_list = list(firsts.keys())
```

```
index = 0
```

```
for gg in firsts:
```

```
    print(f"first({key_list[index]}) ")
```



```
f"=> {firsts.get(gg)}")  
index += 1
```

```
def computeAllFollows():  
    global start_symbol, rules, nonterm_userdef,\  
        term_userdef, diction, firsts, follows  
    for NT in diction:  
        solset = set()  
        sol = follow(NT)  
        if sol is not None:  
            for g in sol:  
                solset.add(g)  
            follows[NT] = solset  
  
    print("\nCalculated follows: ")  
    key_list = list(follows.keys())  
    index = 0  
    for gg in follows:  
        print(f"follow({key_list[index]})"  
            f"=> {follows[gg]}")  
        index += 1
```

```
def createParseTable():  
    import copy  
    global diction, firsts, follows, term_userdef  
    print("\nFirsts and Follow Result table\n")
```

```
mx_len_first = 0
mx_len_fol = 0
for u in diction:
    k1 = len(str(firsts[u]))
    k2 = len(str(follows[u]))
    if k1 > mx_len_first:
        mx_len_first = k1
    if k2 > mx_len_fol:
        mx_len_fol = k2

print(f"{{:<{10}}}} ")
    f"{{:<{mx_len_first + 5}}}} "
    f"{{:<{mx_len_fol + 5}}}} "
    .format("Non-T", "FIRST", "FOLLOW"))
for u in diction:
    print(f"{{:<{10}}}} ")
        f"{{:<{mx_len_first + 5}}}} "
        f"{{:<{mx_len_fol + 5}}}} "
        .format(u, str(firsts[u]), str(follows[u])))

ntlist = list(diction.keys())
terminals = copy.deepcopy(term_userdef)
terminals.append('$')

mat = []
for x in diction:
    row = []
    for y in terminals:
```

```
row.append("")
```

```
mat.append(row)
```

```
grammar_is_LL = True
```

```
for lhs in diction:
```

```
    rhs = diction[lhs]
```

```
    for y in rhs:
```

```
        res = first(y)
```

```
        if '#' in res:
```

```
            if type(res) == str:
```

```
                firstFollow = []
```

```
                fol_op = follows[lhs]
```

```
                if fol_op is str:
```

```
                    firstFollow.append(fol_op)
```

```
                else:
```

```
                    for u in fol_op:
```

```
                        firstFollow.append(u)
```

```
                res = firstFollow
```

```
            else:
```

```
                res.remove('#')
```

```
                res = list(res) + \
```

```
                    list(follows[lhs])
```

```
ttemp = []
```

```
if type(res) is str:
```

```
ttemp.append(res)
res = copy.deepcopy(ttemp)
for c in res:
    xnt = ntlist.index(lhs)
    yt = terminals.index(c)
    if mat[xnt][yt] == "":
        mat[xnt][yt] = mat[xnt][yt] \
            + f"{lhs}->{' '.join(y)}"
    else:
        if f"{lhs}->{y}" in mat[xnt][yt]:
            continue
        else:
            grammar_is_LL = False
            mat[xnt][yt] = mat[xnt][yt] \
                + f",{lhs}->{' '.join(y)}"

print("\nGenerated parsing table:\n")
frmt = "{:>12}" * len(terminals)
print(frmt.format(*terminals))

j = 0
for y in mat:
    frmt1 = "{:>12}" * len(y)
    print(f"{ntlist[j]} {frmt1.format(*y)}")
    j += 1

return (mat, grammar_is_LL, terminals)
```

```
def validateStringUsingStackBuffer(parsing_table, grammarll1,  
                                   table_term_list, input_string,  
                                   term_userdef,start_symbol):
```

```
    print(f"\nValidate String => {input_string}\n")
```

```
    if grammarll1 == False:
```

```
        return f"\nInput String = " \  
               f"\n\"{input_string}\" \n" \  
               f"\nGrammar is not LL(1)"
```

```
    stack = [start_symbol, '$']
```

```
    buffer = []
```

```
    input_string = input_string.split()
```

```
    input_string.reverse()
```

```
    buffer = ['$'] + input_string
```

```
    print("{:>20} {:>20} {:>20}").
```

```
          format("Buffer", "Stack","Action"))
```

```
    while True:
```

```
        if stack == ['$'] and buffer == ['$']:
```

```
            print("{:>20} {:>20} {:>20}")
```

```
.format(' '.join(buffer),
        ' '.join(stack),
        "Valid"))
return "\nValid String!"
elif stack[0] not in term_userdef:

x = list(diction.keys()).index(stack[0])
y = table_term_list.index(buffer[-1])
if parsing_table[x][y] != ":

    entry = parsing_table[x][y]
    print("{:>20} {:>20} {:>25}".
           format(' '.join(buffer),
                   ' '.join(stack),
                   f"T[{stack[0}}][{buffer[-1}}] = {entry}"))
    lhs_rhs = entry.split(">")
    lhs_rhs[1] = lhs_rhs[1].replace('#', "").strip()
    entryrhs = lhs_rhs[1].split()
    stack = entryrhs + stack[1:]
else:
    return f"\nInvalid String! No rule at " \
           f"Table[{stack[0}}][{buffer[-1}}]."
else:

if stack[0] == buffer[-1]:
    print("{:>20} {:>20} {:>20}"
           .format(' '.join(buffer),
                   ' '.join(stack),
                   f"Matched:{stack[0}}"))
    buffer = buffer[:-1]
```

```
stack = stack[1:]  
else:  
    return "\nInvalid String! " \  
        "Unmatched terminal symbols"
```

```
sample_input_string = None
```

```
rules=["S -> A k O",  
       "A -> A d | a B | a C",  
       "C -> c",  
       "B -> b B C | r"]
```

```
nonterm_userdef=['A','B','C']  
term_userdef=['k','O','d','a','c','b','r']  
sample_input_string="a r k O"
```

```
diction = { }  
firsts = { }  
follows = { }
```

```
computeAllFirsts()
```

```
start_symbol = list(diction.keys())[0]
```

```
computeAllFollows()
```

```
(parsing_table, result, tabTerm) = createParseTable()
```

```
if sample_input_string != None:
```


```
    validity = validateStringUsingStackBuffer(parsing_table, result,  
                                                tabTerm, sample_input_string,  
                                                term_userdef,start_symbol)
```

```
    print(validity)
```

```
else:
```

```
    print("\nNo input String detected")
```


Output:

 IDLE Shell 3.9.6

File Edit Shell Debug Options Window Help

Rules:

```
S->[['A', 'k', 'O']]
A->[['A', 'd'], ['a', 'B'], ['a', 'C']]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
```

After elimination of left recursion:

```
S->[['A', 'k', 'O']]
A->[['a', 'B', "A'"], ['a', 'C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]
```

After left factoring:

```
S->[['A', 'k', 'O']]
A->[['a', "A'"]]
A'->[['B', "A'"], ['C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]
```

Calculated firsts:

```
first(S) => {'a'}
first(A) => {'a'}
first(A') => {'c', 'b', 'r'}
first(C) => {'c'}
first(B) => {'b', 'r'}
first(A') => {'#', 'd'}
```

Calculated follows:

```
follow(S) => {'$'}
follow(A) => {'k'}
follow(A') => {'k'}
follow(C) => {'d', 'c', 'k'}
follow(B) => {'d', 'c', 'k'}
follow(A') => {'k'}
```

Firsts and Follow Result table

IDLE Shell 3.9.6

File Edit Shell Debug Options Window Help

```
follow(A) => {'k'}
follow(A'') => {'k'}
follow(C) => {'d', 'c', 'k'}
follow(B) => {'d', 'c', 'k'}
follow(A') => {'k'}
```

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	{ 'a' }	{ '\$' }
A	{ 'a' }	{ 'k' }
A''	{ 'c', 'b', 'r' }	{ 'k' }
C	{ 'c' }	{ 'd', 'c', 'k' }
B	{ 'b', 'r' }	{ 'd', 'c', 'k' }
A'	{ '#', 'd' }	{ 'k' }

Generated parsing table:

	k	O	d	a	c	b	r	\$
S				S->A k O				
A				A->a A''				
A''					A''->C A'	A''->B A'	A''->B A'	
C					C->c			
B						B->b B C	B->r	
A'	A'->#		A'->d A'					

Validate String => a r k O

Buffer	Stack	Action
\$ O k r a	S \$	T[S][a] = S->A k O
\$ O k r a	A k O \$	T[A][a] = A->a A''
\$ O k r a	a A'' k O \$	Matched:a
\$ O k r	A'' k O \$	T[A''] [r] = A''->B A'
\$ O k r	B A' k O \$	T[B][r] = B->r
\$ O k r	r A' k O \$	Matched:r
\$ O k	A' k O \$	T[A'] [k] = A'->#
\$ O k	k O \$	Matched:k
\$ O	O \$	Matched:O
\$	\$	Valid

Valid String!

>>>