WEEK-5: LL(1) Parser:

Implement LL(1) parser:

```
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
```

→ import first & follow set codes

```
def computeAllFirsts():
        global rules, nonterm_userdef, term_userdef,
               diction, firsts.
        for rule in rules:
            k = rule.split("→").
            k[0] = k[0].strip()
            k[1] = k[1].strip()
            rhs = k[1]
            multirhs = rhs.split('|')
```

```
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split().
    diction[k[0]] = multirhs.
print("\n Rules:\n").
for y in diction:
    print("{y} →{diction[y]}").
print("After elimination of left recursion:\n").
diction = removeleftRecursion(diction)
for y in diction:
    print("{y} →{diction[y]}")
print("\n After left factoring:\n").
diction = ~~rem~~ LeftFactoring(diction).
for y in diction:
    print("{y} →{diction[y]}")
~~print("A~~.
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("Calculated first "); key_list = list(firsts.keys())
index = 0
for gg in firsts:
    print("first({key_list[index]}) " ⇒{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("Calculated follows:").
key_list = list(follows.keys()); index = 0;
for gg in follows: print("follow({key_list[index]})
                    ⇒{follows[gg]}").
    index += 1.
```

```
def createParseTable():
    import copy.
    global diction, firsts, follows, term_userdef.
    print("\n First & 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("{{:<{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:
                    firstFdlow.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] + {lhs}->{join(y)}"
    else: if "{lhs}->{y}" in mat[xnt][yt]:
                continue
          else: grammar_is_LL = False.
                mat[xnt][yt] = mat[xnt][yt] \ + {lhs}->{'join(y)}
return (mat, grammar_isLL, terminals).
def validateStringUsingStackBuffer(parsing_table, grammarLL1,
    {..}.                                table_term_list):
if sample_input_string != None:
    validity = validateStringUsingStackBuffer(...)
```

Rules:

S → [['A', 'k', 'o']]

A → [['A', 'd'], ['a', 'B'], ['a', 'c']]

C → [['c']]

B → [['b', 'B', 'c'], ['r']].

OUTPUT:

Firsts and Follow Result Table:

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

Parsing Table:

	k	o	d	a	c	b	r	$
S				S→AkO				
A				A→aA''				
A''					A''→CA'	A''→BA'	A''→BA'	
C					C→c			
B						B→bBC	B→r	
A'			A'→dA'					

A'→#

Validate String ⇒ ark o.

Valid String!