Shri Ramdeobaba College of Engineering and Management, Nagpur
Department of Computer Science and Engineering
Session: 2021-2022 [EVEN SEM]

Compiler Design Lab

Name : Janak Mandavgade

Sec : A

Roll no. : 43

Batch : A3

Subject : Compiler Design

PRACTICAL No. 4

Aim:

(A) Write a program to validate a natural language sentence. Design a natural language grammar, compute and input the LL (1) table. Validate if the given sentence is valid or not based on the grammar.

(B) Use Virtual Lab on LL1 parser to validate the string and verify your string validation using simulation.

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("->")
        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(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"\"{input_string}\"\n" \
               f"Grammar 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 -> NP VP",
         "NP -> P | PN | D N",
         "VP -> V NP",
         "N -> championship | ball | toss",
         "V -> is | want | won | played",
         "P -> me | I | you",
         "PN -> India | Australia | Steve | John",
         "D -> the | a | an"]

nonterm_userdef = ['S', 'NP', 'VP', 'N', 'V', 'P', 'PN', 'D']
term_userdef = ["championship", "ball", "toss", "is", "want",
                "won", "played", "me", "I", "you", "India",
                "Australia","Steve", "John", "the", "a", "an"]
sample_input_string = "India won the championship"

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")
```

Input:

```
["S -> NP VP",

        "NP -> P | PN | D N",

        "VP -> V NP",

        "N -> championship | ball | toss",

        "V -> is | want | won | played",

        "P -> me | I | you",

        "PN -> India | Australia | Steve | John",

        "D -> the | a | an"]
```

Output:

Nonterminal	Nullable?	First	Follow
S	×	me, I, you, India, Australia, Steve, John, the, a, an	
X	×	me, I, you, India, Australia, Steve, John, the, a, an	$
NP	×	me, I, you, India, Australia, Steve, John, the, a, an	is, want, won, played, $
VP	×	is, want, won, played	$
N	×	championship, ball, toss	is, want, won, played, $
V	×	is, want, won, played	me, I, you, India, Australia, Steve, John, the, a, an
P	×	me, I, you	is, want, won, played, $
PN	×	India, Australia, Steve, John	is, want, won, played, $
D	×	the, a, an	championship, ball, toss

	$	championship	ball	toss	is	want	won	played	me	I	you	India	Australia	Steve	John	the	a	an
S									S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $	S ::= X $
X									X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP	X ::= NP VP
NP									NP ::= P	NP ::= P	NP ::= P	NP ::= PN	NP ::= PN	NP ::= PN	NP ::= PN	NP ::= D N	NP ::= D N	NP ::= D N
VP					VP ::= V NP	VP ::= V NP	VP ::= V NP	VP ::= V NP										
N		N ::= championship	N ::= ball	N ::= toss														
V					V ::= is	V ::= want	V ::= won	V ::= played										
P									P ::= me	P ::= I	P ::= you							
PN												PN ::= India	PN ::= Australia	PN ::= Steve	PN ::= John			
D																D ::= the	D ::= a	D ::= an

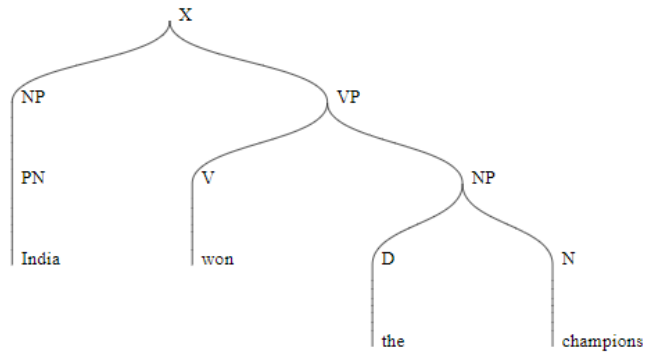
Stack

Remaining Input

Rule

Match $

Partial Parse Tree



```
Validate String => India won the championship

        Buffer                Stack              Action
$ championship the won India                S $    T[S][India] = S->NP VP
$ championship the won India             NP VP $     T[NP][India] = NP->PN
$ championship the won India             PN VP $  T[PN][India] = PN->India
$ championship the won India          India VP $        Matched:India
$ championship the won                   VP $     T[VP][won] = VP->V NP
$ championship the won                 V NP $        T[V][won] = V->won
$ championship the won               won NP $          Matched:won
  $ championship the                   NP $      T[NP][the] = NP->D N
  $ championship the                 D N $        T[D][the] = D->the
  $ championship the               the N $          Matched:the
    $ championship                   N $ T[N][championship] = N->championship
    $ championship       championship $ Matched:championship
                  $                  $               Valid

Valid String!
```