4th March 2022

# Experiment 6: Construction of Predictive Parsing Table

**Aim:** To construct the predictive parsing table and validate an input string for a given grammar.

**Algorithm:**

1. Start.
2. Get input list of terminals and non-terminals from the user
3. Get input list of production rules from the user where # denotes epsilon
4. Eliminate left recursion and left factoring using previous lab algorithms.
5. Generate list of FIRST and FOLLOW using previous lab algorithms.
6. Create a ~~mdr~~ matrix of size (number of non terminals × number of terminals + 1) to store predictive parsing table.
7. For every rule, take lhs and rhs (For E → ab, lhs = E and rhs = ab)
8. For every symbol in rhs:
   - Get FIRST of symbol and store in res
   - If FIRST contains epsilon (#), take union with FOLLOW and store in res.

9. For every character in res, find corresponding matrix position and place rule in matrix (Rule will be lhs → symbol)
10. To parse input, create a stack with starting symbol and # sign and buffer
11. Reverse input string and store in buffer.
12. Loop through the input string.
13. Take front of buffer and stack and look up corresponding rule in predictive parsing table.
14. Append rhs of rule to stack
15. Pop ~~input~~ buffer if stack symbol and buffer symbol match.
16. Repeat steps 13, 14, 15 till all symbols match or buffer empties.
17. If only # is left in both stack and buffer, then input is valid otherwise it is invalid.

# MANUAL WORKING

Grammar :

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Working :

1. Eliminating left recursion and left factoring

$$E \rightarrow E' \mid T$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow T' \mid F$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

2. Calculate FIRST

$$FIRST(E) = \{(, id\}$$
$$FIRST(E') = \{+, \epsilon\}$$
$$FIRST(T) = \{(, id\}$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(F) = \{(, id\}$$

3. Calculate FOLLOW

FOLLOW(E) = { $, ) }
FOLLOW(E') = { $, ) }
FOLLOW(T) = { $, +, ) }
FOLLOW(T') = { $, +, ) }
FOLLOW(F) = { $, *, +, ) }

4. Construction of table

| NTs | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|
| E | | | E → TE' | | E → TE' | |
| E' | E' → +TE' | | | E' → ε | | E' → ε |
| T | | | T → FT' | | T → FT' | |
| T' | T' → ε | T' → *FT' | | T' → ε | | T' → ε |
| F | | | F → (E) | | F → id | |

Rules for table.

For A → α

i) M[A, a] = A → α for a in FIRST(α)
ii) M[A, b] = A → α if ε in FIRST(α), b in FOLLOW(A)

5. Input Parsing

| Buffer | Stack | Action |
|---|---|---|
| $ id + id * id | E $ | E → T E' |
| $ id + id * id | T E' $ | T → F T' |
| $ id + id * id | F T' E' $ | F → id |
| $ id + id * id | id T' E' $ | Pop id |
| $ id + id * | T' E' $ | T' → * F T' |
| $ id + id * | * F T' E' $ | Pop * |
| $ id + id | F T' E' $ | F → id |
| $ id + id | id T' E' $ | Pop id |
| $ id + | T' E' $ | T' → ∈ |
| $ id + | E' $ | E' → + T E' |
| $ id + | + T E' $ | Pop + |
| $ id | T E' $ | T → F T' |
| $ id | F T' E' $ | F → id |
| $ id | id T' E' $ | Pop id |
| $ | T' E' $ | T' → ∈ |
| $ | E' $ | E' → ∈ |
| $ | $ | Valid |

Sashrika Surya
RA1911027010092
Section: N2
Lab Batch: 1
Date: 4th March 2022

# Compiler Design Lab
# Experiment 6: Construction of Predictive Parsing Table

## 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
```

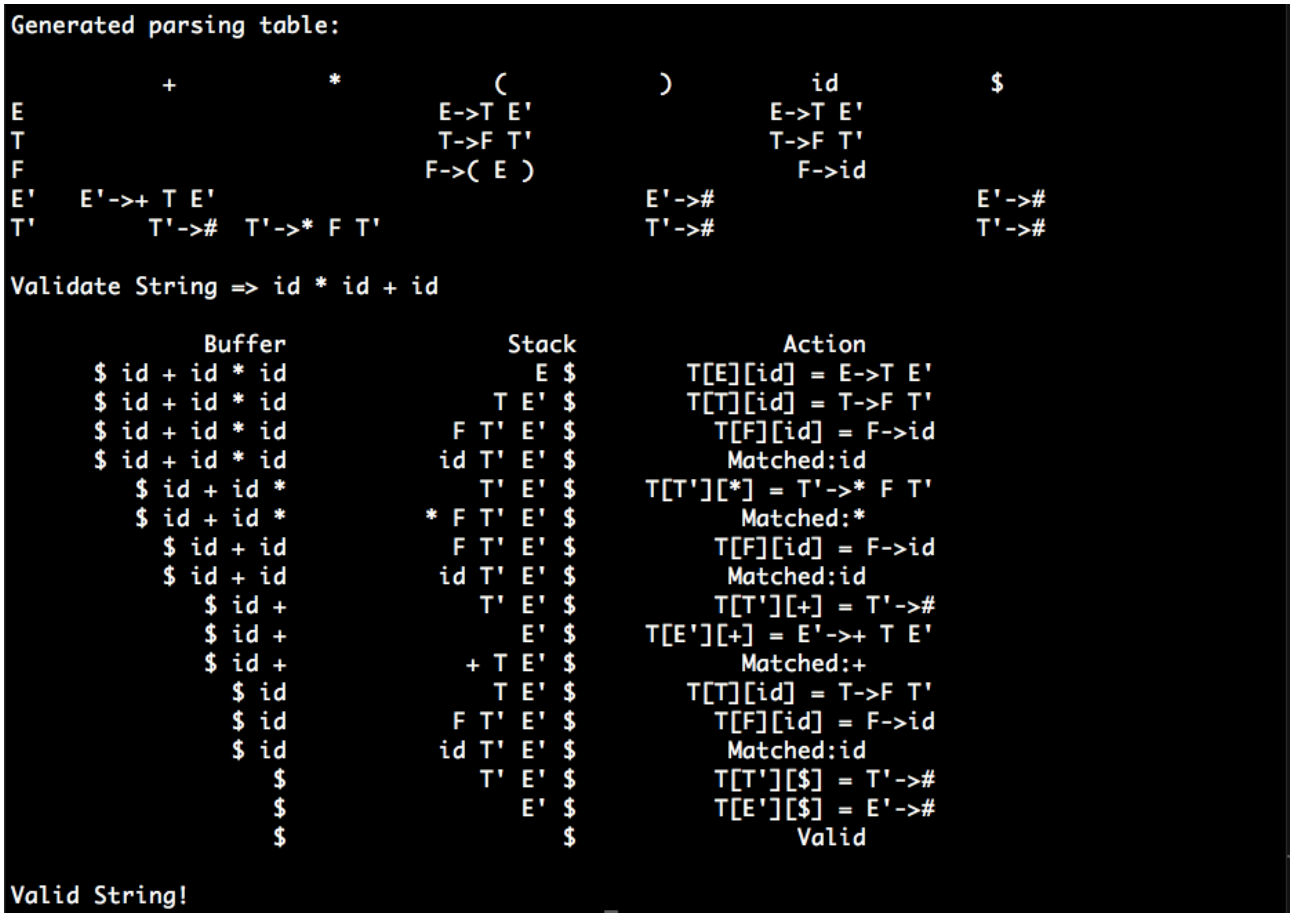
```
    return (mat, grammar_is_LL, terminals)

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 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"

diction = {}
firsts = {}
follows = {}
term_userdef = []
nonterm_userdef = []
rules = []

# Terminals
n1=int(input("Enter no. of terminals: "))
print("Enter terminals:")
for _ in range(n1):
    term_userdef.append(input())

# Non Terminals
n2=int(input("Enter no. of non terminals: "))
print("Enter non terminals:")
for _ in range(n2):
    nonterm_userdef.append(input())

# Production
n3 = int(input("Enter no of productions: "))
print("Enter productions (Sample input: A -> B | c d E) (Epsilon is #):")
for _ in range(n3):
    rules.append(input())

computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()
sample_input_string="id * id + id"

(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**:

```
(base) Sashrikaslaptop:lab6 sashrikasurya$ python lab6.py
Enter no. of terminals: 5
Enter terminals:
+
*
(
)
id
Enter no. of non terminals: 3
Enter non terminals:
E
T
F
Enter starting symbol: E
Enter no of productions: 3
Enter productions (Sample input: A->B|cdE) (Epsilon is @):
E -> E + T | T
T -> T * F | F
F -> ( E ) | id

Rules:

E->[['E', '+', 'T'], ['T']]
T->[['T', '*', 'F'], ['F']]
F->[['(', 'E', ')'], ['id']]

After elimination of left recursion:

E->[['T', "E'"]]
T->[['F', "T'"]]
F->[['(', 'E', ')'], ['id']]
E'->[['+', 'T', "E'"], ['#']]
T'->[['*', 'F', "T'"], ['#']]
```

```
lab6 — -bash — 92×36

After left factoring:

E->[['T', "E'"]]
T->[['F', "T'"]]
F->[['(', 'E', ')'], ['id']]
E'->[['+', 'T', "E'"], ['#']]
T'->[['*', 'F', "T'"], ['#']]

Calculated firsts:
first(E) => {'(', 'id'}
first(T) => {'(', 'id'}
first(F) => {'(', 'id'}
first(E') => {'+', '#'}
first(T') => {'*', '#'}

Calculated follows:
follow(E) => {'$', ')'}
follow(T) => {'$', ')', '+'}
follow(F) => {'$', ')', '+', '*'}
follow(E') => {'$', ')'}
follow(T') => {'$', ')', '+'}

Firsts and Follow Result table

Non-T      FIRST           FOLLOW
E          {'(', 'id'}     {'$', ')'}
T          {'(', 'id'}     {'$', ')', '+'}
F          {'(', 'id'}     {'$', ')', '+', '*'}
E'         {'+', '#'}      {'$', ')'}
T'         {'*', '#'}      {'$', ')', '+'}
```




**Result**:
Hence, the predictive parsing table was constructed and an input string was validated using the table.