# COMPILER DESIGN

## EXP 6 – Predictive parsing table

**Tamojit Sarkar**
**RA1811027010034**
**CSE-BD Sec-I2**

**Aim:** To prepare the predictive parsing table of the grammar.

**Language Used: Python**

**Procedure:**

1. Create a python file
2. Take in the input of statements present in the grammar
3. Find the first and follow of each non-terminal.
4. Prepare the parsing table matching the first and follow and write all the table entries.
5. Then print the table with first and follow and also mention all the entries in the predictive parsing table.

**Code:**

```
#for printing terminals, non_terminals and their entries in Parsing_table
def print_(table):
    print_rules(rules);
    print_FFset(firstSet,True);
    print_FFset(followSet,False);
    terminals=set()
    for nt in rules.keys():
        terminals=terminals.union(firstSet[nt])
        terminals=terminals.union(followSet[nt])
    terminals.discard('eps')
    print("\nNon Terminals:\n",rules.keys())
    print("\nTerminals:\n",terminals)
    print("\n\nTable entries are:\n")
    for row,col in table.items():
        print(row,':',col)
    print()
    return


def get_parsing_table(firstSet,followSet,rules):
    parsing_table=defaultdict()
    table=defaultdict()   #just for printing in good way (can be done by parsing_table too)
    for key,rule in rules.items():
        for sub_rule in rule:
            symbol = sub_rule[0]
            if isNonTerminal(symbol):
                for ter in firstSet[symbol]-{'eps'}:
                    parsing_table[key,ter]={key:sub_rule}
                    table[key,ter]=key+'-> '+' '.join(i for i in sub_rule)

            elif symbol=="eps" or symbol in deepcopy(firstSet[symbol]):
                for ter in followSet[key]:
                    parsing_table[key,ter] = {key:['eps']}
                    table[key,ter]=key+'-> '+'eps'
            else:
                parsing_table[key,symbol]={key:sub_rule}
                table[key,symbol]=key+'-> '+' '.join(i for i in sub_rule)
    print_(table)  #for printing terminals, non_terminals and their entries in Parsing_table
    return parsing_table
```

```
[ ]
    def parser(p_table,start_state):
        expr = list(map(str,input("Enter expression for prasing(Plz enter space between 2 entry)\n").split()))
        if expr[-1] != '$':
            print("\nPlease add '$' at the end of expression. Try again")
            return
        print("\nyour expression is",expr)
        stack=['$'];stack.append(start_state)
        inp=0
        while(stack and expr[inp]):
            popped = stack.pop()
            while (popped=='eps'):  #when popped is epsilon then again pop
                popped = stack.pop()
            if popped != expr[inp]:
                if p_table.get((popped,expr[inp])): #for checking, this entry is in table or not ?
                    rule = p_table.get((popped,expr[inp])).get(popped)  # 2D dict table is again 1D dict with that rule
                    for x in range(len(rule)):
                        stack.append(rule[-x-1])     # minus for reversing
                else:
                    print("\nError, the expression is wrong. Try again")
                    return
            else:
                inp+=1
            if stack[0]==expr[inp]:
                flag=True
                break

        if flag:
            print("\nExpression accepted")
        else:
            print("\nExpression rejected, it can't be generated from this grammar")
        return
```

```
    def get_rules():
        dic = defaultdict()
        start_state='E'
        dic={       #rules of grammar in LL(1) form
            "E" : [ ["T","E1"] ],
            "T" : [ ["F","T1"] ],
            "F" : [ ["id"], ["(","E",")"] ],
            "E1": [ ["+","T","E1"], ["-","T","E1"], ["eps"] ],
            "T1": [ ["*","F","T1"], ["/","F","T1"], ["eps"], ["^","F","T1"] ]
        }
        return dic,start_state


    def print_rules(rules):
        print("\nRules are:")
        for key,rule in rules.items():
            print(key,end=" => ")
            for sub_rule in rule:
                for symbol in sub_rule:
                    print(symbol,end=" ")
                print(' | ',end='')
            print()
        return


    def isNonTerminal(symbol):
        if symbol in rules.keys():#all keys of rules are Non terminals
            return True
        return False
```

```
[ ]  def first(rules,key,firstSet):
        for rule in rules[key]:#for every rule of that Non-terminal
            symbol=rule[0]
            if not isNonTerminal(symbol): #for terminals (not False => True)
                firstSet[key].add(symbol)
            else:                         # for terminal
                firstSet[key]=firstSet[key].union(first(rules,symbol,firstSet))
        return firstSet[key]


    def get_first_set(rules):#rules is defaultdict
        firstSet=defaultdict(set)
        for key in rules.keys():
            firstSet[key]
        for key,rule in rules.items():#for all rules
            first(rules,key,firstSet)
        return firstSet


    def print_FFset(ffset,flag):#both first and follow as FFset
        if flag:
            print("\nFirst sets are:")
        else:
            print("\nFollow sets are:")
        for key,value in ffset.items():
            print(key,"=",value)
        return
```

```
def follow(rules,firstSet,followSet,non_ter):
    # for case; any_non_ter => alpha non_ter beta ( alpha,beta belongsto NonTer or Ter )
    for new_non_ter,rule in rules.items(): #for all productions, 1-1 production
        for sub_rule in rule:  #for one production_rules, 1-1 rule
            for i in range(len(sub_rule)): #for one rule_symbols, 1-1 symbol
                if sub_rule[i]==non_ter:
                    if i+1 < len(sub_rule):#checking next symbol beta
                        beta=sub_rule[i+1]# when beta is present
#                         print("Y => alpha X beta as",new_non_ter,'=>..',non_ter,beta)

                        if isNonTerminal(beta): #add firstSet(beta) into followSet(non_ter)
                            followSet[non_ter]=followSet[non_ter].union(firstSet[beta])
                            followSet[non_ter].discard('eps') #trying to remove epsilon, if there is
                        else:
                            followSet[non_ter].add(beta)

                        if 'eps' in firstSet[beta] and new_non_ter!=beta :#!= for avoiding infinite loop (for T; P=> *TP)
                            followSet[non_ter]=followSet[non_ter].union(follow(rules,firstSet,followSet,new_non_ter))#add firstSet(beta) into followSet(non_ter)

                    elif i+1 == len(sub_rule) and new_non_ter!=non_ter: # when beta is not there & != for avoiding infinite loop
#                         print("Y => alpha X as",new_non_ter,'=>..',non_ter)
                        if isNonTerminal(sub_rule[i]):
                            followSet[non_ter]=followSet[non_ter].union(follow(rules,firstSet,followSet,new_non_ter))

    return followSet[non_ter]


def get_follow_set(rules,firstSet,start_state):#rules and follow are defaultdicts.
    followSet=defaultdict(set)
    for non_ter in rules.keys():
        if non_ter == start_state: #add terminal symbol
            followSet[non_ter].add('$')
        else:
            followSet[non_ter]

    for non_ter in rules.keys():
        follow(rules,firstSet,followSet,non_ter)
    return followSet
```

```
[ ]  #import First_Follow_sets as ffs
     from collections import defaultdict
     from copy import deepcopy
     if __name__=="__main__":

         rules,start_state=get_rules()
         firstSet = get_first_set(rules)
         followSet = get_follow_set(rules,deepcopy(firstSet),start_state)
         parsing_table = get_parsing_table(deepcopy(firstSet),deepcopy(followSet),rules)
        # parser(parsing_table,start_state)
     #     print(parsing_table)
```

## Output:

```
Rules are:
E => T E1  |
T => F T1  |
F => id  | ( E )  |
E1 => + T E1  | - T E1  | eps  |
T1 => * F T1  | / F T1  | eps  | ^ F T1  |

First sets are:
E = {'(', 'id'}
T = {'(', 'id'}
F = {'(', 'id'}
E1 = {'+', 'eps', '-'}
T1 = {'*', '/', '^', 'eps'}

Follow sets are:
E = {'$', ')'}
T = {'-', '+', '$', ')'}
F = {'-', '/', '$', '^', '*', '+', ')'}
E1 = {'$', ')'}
T1 = {'+', ')', '$', '-'}

Non Terminals:
 dict_keys(['E', 'T', 'F', 'E1', 'T1'])

Terminals:
 {'-', '(', '/', '$', '^', '*', '+', 'id', ')'}


Table entries are:

('E', '(') : E-> T E1
('E', 'id') : E-> T E1
('T', '(') : T-> F T1
('T', 'id') : T-> F T1
('F', 'id') : F-> id
('F', '(') : F-> ( E )
('E1', '+') : E1-> + T E1
('E1', '-') : E1-> - T E1
('E1', '$') : E1-> eps
('E1', ')') : E1-> eps
('T1', '*') : T1-> * F T1
('T1', '/') : T1-> / F T1
('T1', '-') : T1-> eps
('T1', '+') : T1-> eps
('T1', '$') : T1-> eps
('T1', ')') : T1-> eps
('T1', '^') : T1-> ^ F T1
```

**Conclusion:** **Predictive parsing table for the grammar is prepared.**