Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 24th September 2021

Aim

To design predictive parser from given grammar. (LL1)

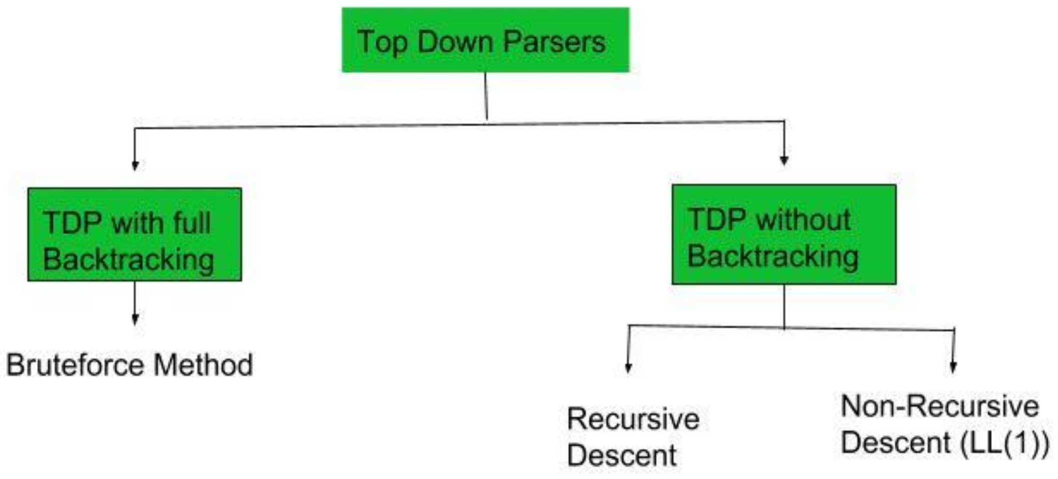
Program logic

1. First and Follow Sets
2. LL1 parsing Table
3. String parsing function which takes string as Input and outputs whether the string is accepted or rejected by the grammar

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



LL (1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL (1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar (or parsing table) or not.
Let given grammar is G = (V, T, S, P)
where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production set.

LL (1) Parser algorithm:

Input

1. stack = S //stack initially contains only S.

2. input string = w$
   where S is the start symbol of grammar, w is given string, and $ is used for the end of string.
3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output

Determines that given string can be produced by given grammar (parsing table) or not, if not then it produces an error.

Steps

```
1. while(stack is not empty) {

       // initially it is S
2.     A = top symbol of stack;

       //initially it is first symbol in string, it can be $ also
3.     r = next input symbol of given string;
4.     if (A∈T or A==$) {
5.         if(A==r){
6.             pop A from stack;
7.             remove r from input;
8.         }
9.         else
10.             ERROR();
11.     }
12.     else if (A∈V) {
13.         if(PT[A,r]= A⇢B1B2....Bk) {
14.             pop A from stack;

                // B1 on top of stack at final of this step
15.             push Bk,Bk-1......B1 on stack
16.         }
17.         else if (PT[A,r] = error())
18.             error();
19.     }
20. }
```

// if parser terminate without error ()

// then given string can generated by given parsing table.

p is the maximum of lengths of strings in RHS of all productions and

l is the length of given string and

l is very larger than p. if block at line 4 of algorithm always runs for O(1) time. else if block at line 12 in algorithm takes O(|P|*p) as upper bound for a single next input symbol. And while loop can run for more than l times, but we have considered the repeated while loop for a single next input symbol in O(|P|*p). So, the total time complexity is

T(l) = O(l)*O(|P|*p)

= O(l*|P|*p)

= O(l) { as l >>>|P|*p }

The time complexity of this algorithm is the order of length of the input string.

Comparison with Context-free language (CFL):

Languages in LL(1) grammar is a proper subset of CFL. Using the CYK algorithm we can find membership of a string for a given Context-free grammar (CFG). CYK takes O(l3) time for the membership test for CFG. But for LL(1) grammar we can do a membership test in O(l) time which is linear using the above algorithm. If we know that given CFG is LL(1) grammar then use LL(1) parser for parsing rather than CYK algorithm.

Time complexity

As we know that size of a grammar for a language is finite. Like, a finite number of variables, terminals, and productions. If grammar is finite then its LL(1) parsing table is also finite of size O(V*T). Let

Example

Let the grammar G = (V, T, S', P) is

*S' → **S$***
S → xYzS | a
Y → xYz | y

Parsing table (PT) for this grammar

	a	**x**	**y**	**z**	**$**
S'	S' → S$	S' → S$	error	error	error
S	S → a	S → xYzS	error	error	error
Y	error	Y → xYz	Y → y	error	error

Let *string1* = *xxyzza*,

We have to add $ with this string,
We will use the above parsing algorithm, diagram for the process:

For string1 we got an empty stack, and while loop or algorithm terminated without error.
So, string1 belongs to language for given grammar G.

Let $string2 = xxyzzz$,

For string2, at the last stage as in the above diagram when the top of the stack is S and the next input symbol of the string is z, but in PT[S,z] = error. The algorithm terminated with an error. So, string2 is not in the language of grammar G.

First Function-

First(α) is a set of terminal symbols that begin in strings derived from α.

Example-

Consider the production rule-

A → abc / def / ghi

Then, we have-

First(A) = {a, d, g}

Rules For Calculating First Function-

Rule-01:

For a production rule X → ∈,

First(X) = {∈ }

Rule-02:

For any terminal symbol ‘a’,

First(a) = { a }

Rule-03:

For a production rule X → Y1Y2Y3,

Calculating First(X)

- If ∈ ∉ First(Y1), then First(X) = First(Y1)
- If ∈ ∈ First(Y1), then First(X) = { First(Y1) – ∈ } ∪ First(Y2Y3)

Calculating First(Y2Y3)

- If ∈ ∉ First(Y2), then First(Y2Y3) = First(Y2)
- If ∈ ∈ First(Y2), then First(Y2Y3) = { First(Y2) – ∈ } ∪ First(Y3)

Similarly, we can make expansion for any production rule X → Y1Y2Y3…..Yn.

Follow Function-

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule A → αB,

Follow(B) = Follow(A)

Rule-03:

For any production rule A → αBβ,

- If ∈ ∉ First(β), then Follow(B) = First(β)
- If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Important Notes-

Note-01:

- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.

Note-02:

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

Note-03:

Varun Khadayate
A016

We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Practice problems based on calculating first and follow-

Problem-01:

Calculate the first and follow functions for the given grammar-

$$S \rightarrow aBDh$$

$$B \rightarrow cC$$

$$C \rightarrow bC \,/\, \in$$

$$D \rightarrow EF$$

$$E \rightarrow g \,/\, \in$$

$$F \rightarrow f \,/\, \in$$

Solution-

The first and follow functions are as follows-

First Functions-

- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow Functions-

- Follow(S) = {$ }
- Follow(B) = {First(D) – ∈} ∪ First(h) = {g , f , h }
- Follow(C) = Follow(B) = {g, f , h }
- Follow(D) = First(h) = {h }
- Follow(E) = {First(F) – ∈} ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = {h }

LAB Assignment

What is LL1 Parser

A top-down parser that uses a one-token lookahead is called an LL(1) parser.

- The first L indicates that the input is read from left to right.
- The second L says that it produces a left-to-right derivation.
- And the 1 says that it uses one lookahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.)

What is First and Follow?

First(α) is a set of terminal symbols that begin in strings derived from α.

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Specify rules for LL1 parser.

Specify rules for first and follow.

Rules For Calculating First Function-

Rule-01:

For a production rule $X \to \in$,

$$\text{First}(X) = \{ \in \}$$

Rule-02:

For any terminal symbol 'a',

$$\text{First}(a) = \{ a \}$$

Rule-03:

For a production rule $X \to Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin \text{First}(Y_1)$, then $\text{First}(X) = \text{First}(Y_1)$
- If $\in \in \text{First}(Y_1)$, then $\text{First}(X) = \{ \text{First}(Y_1) - \in \} \cup \text{First}(Y_2Y_3)$

Calculating First(Y2Y3)

- If $\in \notin \text{First}(Y_2)$, then $\text{First}(Y_2Y_3) = \text{First}(Y_2)$
- If $\in \in \text{First}(Y_2)$, then $\text{First}(Y_2Y_3) = \{ \text{First}(Y_2) - \in \} \cup \text{First}(Y_3)$

Similarly, we can make expansion for any production rule $X \to Y_1Y_2Y_3.....Y_n$.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place \$ in Follow(S).

Rule-02:

For any production rule $A \to \alpha B$,

$$\text{Follow}(B) = \text{Follow}(A)$$

Rule-03:

For any production rule $A \to \alpha B\beta$,

- If $\in \notin \text{First}(\beta)$, then $\text{Follow}(B) = \text{First}(\beta)$
- If $\in \in \text{First}(\beta)$, then $\text{Follow}(B) = \{ \text{First}(\beta) - \in \} \cup \text{Follow}(A)$

Define algorithm for LL1parser.

Input

1. stack = S //stack initially contains only S.
2. input string = w$
   where S is the start symbol of grammar, w is given string, and $ is used for the end of string.
3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output

Determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

```
1. while(stack is not empty) {

       // initially it is S
2.     A = top symbol of stack;

       //initially it is first symbol in string, it can be $ also
3.     r = next input symbol of given string;
4.     if (A∈T or A==$) {
5.         if(A==r){
6.             pop A from stack;
7.             remove r from input;
8.         }
9.         else
10.             ERROR();
11.     }
12.     else if (A∈V) {
13.         if(PT[A,r]= A⇢B1B2....Bk) {
14.             pop A from stack;

                // B1 on top of stack at final of this step
15.             push Bk,Bk-1......B1 on stack
16.         }
17.         else if (PT[A,r] = error())
18.             error();
```

```
    }
}
```

Lab Assignment Program

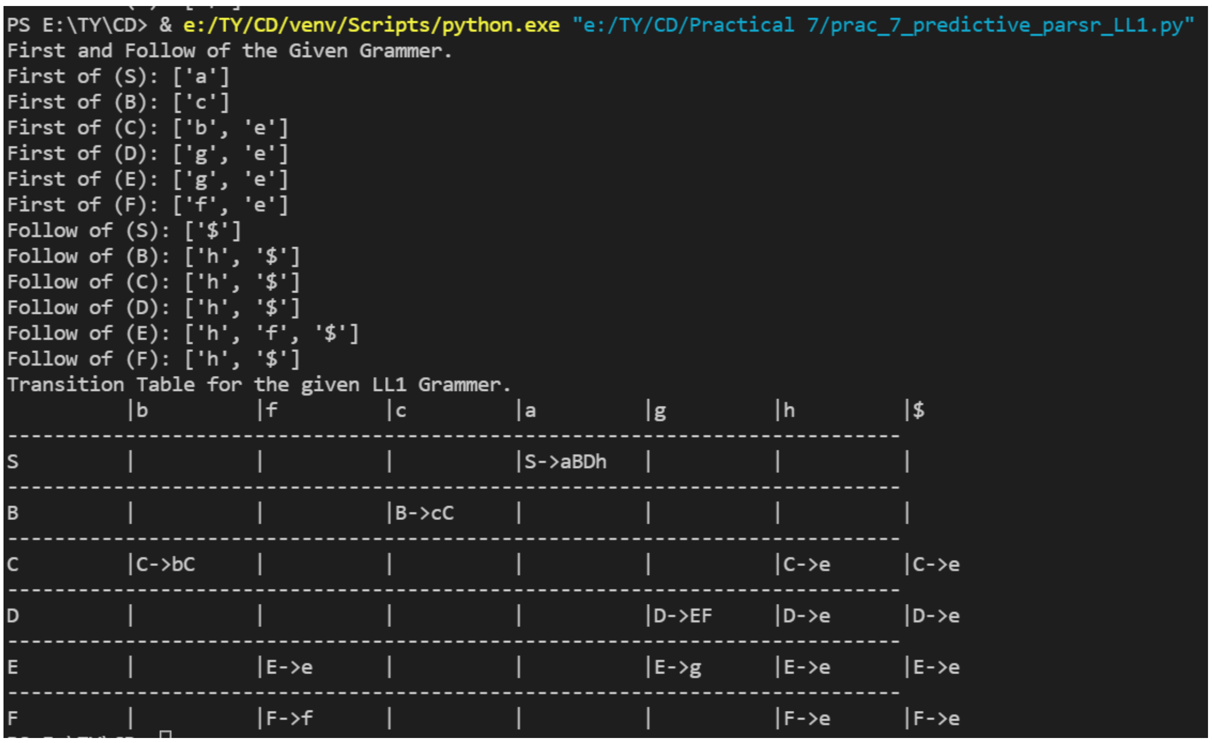
Write a program to design predictive parser from given grammar. (LL1)

Code

```python
gram = {
    "S":["aBDh"],
    "B":["cC"],
    "C":["bC","e"],
    "D":["EF"],
    "E":["g","e"],
    "F":["f","e"]
}

def removeDirectLR(gramA, A):
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            tempInCr.append(i[1:]+[A+"'"])
        else:
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA

def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)
        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)

    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

result = rem(gram)
terminals = []
for i in result:
    for j in result[i]:
        for k in j:
            if k not in result:
                terminals+=[k]
terminals = list(set(terminals))

def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

print("First and Follow of the Given Grammer.")
firsts = {}
for i in result:
    firsts[i] = first(result,i)
    print(f'First of ({i}):',firsts[i])

def follow(gram, term):
    a = []
```

```python
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
print("\n")
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
    print(f'Follow of ({i}):',follows[i])

resMod = {}
for i in result:
    l = []
    for j in result[i]:
        temp = ""
        for k in j:
            temp+=k
        l.append(temp)
    resMod[i] = l

print("Transition Table for the given LL1 Grammer.")

tterm = list(terminals)
tterm.pop(tterm.index("e"))
tterm+=["$"]
pptable = {}
for i in result:
    for j in tterm:
        if j in firsts[i]:
            pptable[(i,j)]=resMod[i[0]][0]
        else:
            pptable[(i,j)]=""
    if "e" in firsts[i]:
        for j in tterm:
            if j in follows[i]:
                pptable[(i,j)]="e"
pptable[("F","i")] = "i"
toprint = f'{"": <10}'
for i in tterm:
    toprint+= f'|{i: <10}'
print(toprint)
for i in result:
    toprint = f'{i: <10}'
    for j in tterm:
        if pptable[(i,j)]!="":
            toprint+=f'|{i+"->"+pptable[(i,j)]: <10}'
        else:
            toprint+=f'|{pptable[(i,j)]: <10}'
    print(f'{"-":-<76}')
    print(toprint)
```

Output



```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 7/prac_7_predictive_parsr_LL1.py"
First and Follow of the Given Grammer.
First of (S): ['a']
First of (B): ['c']
First of (C): ['b', 'e']
First of (D): ['g', 'e']
First of (E): ['g', 'e']
First of (F): ['f', 'e']
Follow of (S): ['$']
Follow of (B): ['h', '$']
Follow of (C): ['h', '$']
Follow of (D): ['h', '$']
Follow of (E): ['h', 'f', '$']
Follow of (F): ['h', '$']
Transition Table for the given LL1 Grammer.
          |b         |f         |c         |a         |g         |h         |$
----------------------------------------------------------------------------
S         |          |          |          |S->aBDh   |          |          |
----------------------------------------------------------------------------
B         |          |          |B->cC     |          |          |          |
----------------------------------------------------------------------------
C         |C->bC     |          |          |          |          |C->e      |C->e
----------------------------------------------------------------------------
D         |          |          |          |          |D->EF     |D->e      |D->e
----------------------------------------------------------------------------
E         |          |E->e      |          |          |E->g      |E->e      |E->e
----------------------------------------------------------------------------
F         |          |F->f      |          |          |          |F->e      |F->e
```

Conclusion

Hence, we were able to design a predictive parser from given grammar. (LL1)