Experiment No. 6

Aim : Implementation of LL(1) Parser.

Theory :

LL(1) Parser :

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 whether a 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) Grammer :

The first 'L' in LL(1) stands for scanning the input from left to right, the second 'L' stands for producing a leftmost derivation, and the '1' for using one input symbol of lookahead at each step to make parsing action decisions. LL(1) grammar follows the Top-down parsing method. For a class of grammars called LL(1) we can construct a grammar predictive parser. That works on the concept of recursive-descent parser not requiring backtracking.

Elimination of Left Recursion

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow A\ \alpha \mid \beta$. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. Left recursion can be eliminated by modifying the rule as follows: (A' is new non-terminal and ε is representing epsilon).

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \varepsilon$

Left Factoring

It is a grammar transformation that is useful for producing grammar suitable for predictive or top-down parsing. When the choice between two alternative productions is not clear, we rewrite the productions to defer the decision to make the right choice.

For example, if we have grammar rule A → α β1 | α β2 A

→ α A'

A' → β1 | β2

Concept of First and Follow

The construction of a top-down parser is aided by FIRST and FOLLOW functions, that are associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.

Rules for First computation :

1. If x is terminal, then FIRST(x)={x}
2. If X→ ε is production, then add ε to FIRST(X) 3. If X is a non-terminal and X → PQR then FIRST(X)=FIRST(P)
4. If FIRST(P) contains ε, then

   FIRST(X) = (FIRST(P) – {ε}) U FIRST(QR)

Rules for Follow computation :

Epsilon (ε) can never be present in the FOLLOW of any non-terminal symbol.

1. For Start symbol, place $ in FOLLOW(S)
2. If A→ α B, then FOLLOW(B) = FOLLOW(A)
3. If A→ α B β, then

   If ε not in FIRST(β), FOLLOW(B)

   = FIRST(β)

   else do,

FOLLOW(B) = (FIRST(β)-{ε}) U FOLLOW(A)

Parsing table :

After the construction of the parsing table, if for any non-terminal symbol in the table we have more than one production rule for any terminal symbol in the table column the grammar is not LL(1). Otherwise, then grammar is considered as LL(1).

Rules for construction of parsing table :

Step 1 : For each production A → α , of the given grammar perform Step 2 and Step 3.

Step 2 : For each terminal symbol 'a' in FIRST(α), ADD A → α in table T[A,a], where 'A' determines row & 'a' determines column.

Step 3 : If ε is present in FIRST(α) then find FOLLOW(A), ADD A → ε, at all columns 'b', where 'b' is FOLLOW(A). (T[A,b])

Step 4 : If ε is in FIRST(α) and $ is the FOLLOW(A), ADD A → α to T[A,$]. The assumption made in code :

a. The LHS symbol of the First rule is considered as the start symbol.
b. '#' represents the epsilon symbol.

LL(1) Parser algorithm :
Input

1. stack = S //stack initially contains only S.
   input string = w$
2. 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.

Grammer:

S->ACB|CbB|Ba
A->da|BC
B->g|#
C->h|#

Program:

```
import re
ep = list()
fp = open("/content/grammer.txt","r") #read CFG from file cfg=dict() #create
dictionary to stored CFG global non_terminal
def find_first(key):
 value=cfg[key] #find RHS of key(LHS)
 #print(key,value)
 if ('#' in value): #if key directly derived epsilon  value.remove('#')
 for item in value: #consider individual production rule  if item[0] in ep: #if that
variable produve epsilon  epsilon(item)
 #print("Epsilon called for ",item)  else:
 if (item[0].islower()):
 #print(non_terminal,"-->",item[0])  if item[0] not in temp:
 temp.append(item[0])
 else:
 find_first(item[0])

def epsilon(item):
 #print("From Epsilon ",item[0])
 find_first(item[0]) #find first of that variable  length=len(item)
 i=1
 while(i<=length-1):
 if item[i] in ep:
 find_first(item[i])
 i=i+1
 if(i==length):
 #print(non_terminal,"-->#")
 if '#' not in temp:
 temp.append('#')
 break
 else:
```

```
if (item[i].islower()):
#print(non_terminal,"-->",item[i])
```



```
if item[i] not in temp:
temp.append(item[i])
break
else:
find_first(item[i])
break

def find_follow(key):

for k,v in cfg.items():

for item in v:

if re.search(key,item): #search key in RHS
index=item.find(key) #if found
length=len(item)-1
#print(key ,"is found at ",item, "at ",index)

if (index==length):#if varaibel found at Right most side of  RHS
temp1=follow[k] #then find follow[k]  for i in temp1:
temp.append(i) #append it

index=index+1 #find next symbol
#print(index)
for i in range(index,len(item)): #try all next  variable/terminal
#print(i)
                                                    if(item[i].islower()): #if follow symbol is terminal
temp.append(item[i]) #add it to follow()  break # stop
else:
temp1=first[item[i]] #find first[follow variable]  #print(temp1)
                                        for j in temp1:
if(j!='#'):
temp.append(j) #append it to follow  dictionary/temp

if ('#' in temp1): #if first[follow item] contain  epsilon
i=i+1 #check for next variable  else:
break #else stop
```



```
if(i==len(item)): #if we reach at right most side of RHS  temp1=follow[k] #find follow[LHS]
for j in temp1:
temp.append(j) #append result
```

```
for line in fp:
 line.strip() #remove begining and trailing white spaces  if re.search('\n',line):
 line=line[:line.find('\n')] #remove \n from line  split=line.split('->') #split line
(LHS and RHS)
 split=split[1].split('|') #split RHS based on |
 i=0
 for item in split:
 split[i]=item.strip() #remove begining and trailing white spaces  i=i+1
 #cfg[split[0]]=split
 cfg[line[0]]=split #store LHS as key and RHS as a values


print("\nGiven Context Free Grammar is =")

for key, value in cfg.items():
 print(key," ->",value) #print CFG
 if('#' in value): #if any variable/non terminal generate an epsilon  ep.append(key) #then stored
that varaible in list ep

temp=list()
first=dict()
for key,value in cfg.items():
 first[key]=[] #initialize value of key as list
 #print("FIRST[",key,"]") #find first of all variable(non-terminal)  non_terminal=key
 find_first(key)
 if key in ep:
 #print(non_terminal,"-->#")
 if '#' not in temp: #if varaible produce epsilon  temp.append('#') #add epsilon to
first[variable]  #print(temp)
 for item in temp:
 first[non_terminal].append(item) #add all results to first  #print(temp)
 temp.clear()
```



```
#print(first)
#print("Follow() are as follow->")
follow=dict()
flag=0
temp=list()

#print(first)
```

```
for key,value in cfg.items():
 follow[key]=[] #initialize value of key as list
 if flag==0:
 temp.append("$") #follow of start symbol is $
 flag=1
 find_follow(key)
 #print("follow[",key,"]")
 #print(temp)
 for k in temp:
 if( k not in follow[key]): # removed duplicate and add it to final  result
 follow[key].append(k)
 temp.clear()

#print(follow)

print(" Non Terminal First() Follow()")  print("-----------------------------------------------------")

for key, value in follow.items():
 print(" ",key," ",first[key]," ",value)  print("\n")


print(cfg)
```


Output:

```
Given Context Free Grammar is =
S -> ['ACB', 'CbB', 'Ba']
A -> ['da', 'BC']
B -> ['g', '#']
C -> ['h', '#']
 Non Terminal First() Follow()
---------------------------------------------------   S ['d', 'g', 'h', '#', 'b', 'a'] ['$']
```

A ['d', 'g', 'h', '#'] ['h', 'g', '$'] B ['g', '#'] ['$', 'a', 'h', 'g'] C ['h', '#'] ['g', '$', 'b', 'h']

{'S': ['ACB', 'CbB', 'Ba'], 'A': ['da', 'BC'], 'B': ['g'], 'C': ['h']}

Conclusion:

Thus we have successfully implemented LL(1) Parser.