Name: Bhavesh Kewalramani
Roll No.: A-25

Practical No. 4

Theory

LL(1) Parsing:
Here the 1st **L** represents that the scanning of the Input will be done from Left to Right manner and the second **L** shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:
Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.
Step 2: Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.
   To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Practicals

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 based on the grammar or not.
(B) Use Virtual Lab on LL1 parser to validate the string and verify your string validation using simulation.

Program:

```
from collections import OrderedDict
import re

# ` reprsents EPSILON

def getGrammar():
    terminal=[]
    nonterminal=[]
    start=""
    rule=dict()
    flag=0
    print("Enter the Production Rules : ")
    while(True):
        inp=input("===> ")
        if(inp==""):
            break
        s1,s2=inp.split("~")
        if flag==0:
            start=s1
            flag=1
        rule[s1]=[]
        s2=list(s2.split("/"))
        for i in s2:
            rule[s1].append(i)

    r=[]
    k=rule.keys()
    for i in rule.values():
        for j in i:
            for a in list(j.split(" ")):
                if a not in k:
                    if a not in r:
                        r.append(a)

    r.append("$")
    t=dict()
    for i in rule.keys():
        t[i]=dict()
        for j in r:
            t[i].update({j:set()})

    return rule,start,r,t

def Calculate_First(s,rule,v):
    if s[0] in v:
        return set([s[0]])
```

```
    elif s[0] == '`':
        return set(["`"])
    else:
        res = set()
        for j in rule[s[0]]:
            h = Calculate_First(list(j.split(' ')),rule,v)
            res.update(set(h))
        if len(s) == 1:
            return res
        else:
            if '`' in res:
                res.remove('`')
                return res.union(Calculate_First(list(s.split(' '))[1:],rule,v))
            return res


def getValue(v,rule):
    for key, value in rule.items():
        if value == v:
            return key


def Calculate_Follow(s,rule,v,start):
    res = set()
    if s == start:
        res = set(['$'])
    for i in rule.values():
        for j in range(len(i)):
            l = list(i[j].split(' '))
            for k in range(len(l)):
                if l[k] == s:
                    if k == len(l) - 1:
                        if getValue(i,rule) == s:
                            continue
                        else:
                            res.update(set(Calculate_Follow(getValue(i,rule),rule,v,start)))
                    else:
                        c = set(Calculate_First(l[k + 1:],rule,v))
                        if '`' in c:
                            c.remove('`')
                            fol = Calculate_Follow(getValue(i,rule),rule,v,start)
                            c.update(fol)
                            res.update(c)
                        else:
                            res.update(set(c))
    return res
```

```
def parseString(string,t):
    print("Given String is : ", string)
    string = list(string.split(' '))
    string.append('$')
    stk = ['$', start]
    print("Input                                    Stack")
    print(string[::-1] , "\t\t\t\t\t", stk[::-1])
    while not len(stk) == 0:
        top = stk[-1]
        stk.pop()
        if string[0] == '$' and top == '$' and len(stk) == 0:
            print("***** String Accepted *****")
        elif (string[0] == '$' and len(stk) != 0) or (string[0] != '$' and len(stk) == 0):
            print("***** String Rejected *****")
        elif top == string[0]:
            string = string[1:]
        else:
            for i in t[top][string[0]]:
                l = list(i.split(' '))
                for j in l[::-1]:
                    stk.append(j)
        print(stk[::-1] , "\t\t\t\t\t", stk[::-1])

if __name__=="__main__":

    d,start,r,t=getGrammar()

    print("===== FIRST =====")
    for i in d.keys():
        print("first(", i, ") : ", Calculate_First([i],d,r))
    print("===== FOLLOW =====")
    for i in d.keys():
        print("follow(", i, ") : ", Calculate_Follow(i,d,r,start))
    print("===== Parsing Table =====")

    for i in d.keys():
        for rule in d[i]:
            f = Calculate_First(list(rule.split(' ')),d,r)
            if '`' in f:
                fol = Calculate_Follow(i,d,r,start)
                for j in fol:
                    if j != '$':
                        t[i][j].add(rule)
                if '$' in fol:
```

```
                t[i]['$'].add(rule)
        else:
            for j in f:
                t[i][j].add(rule)
print("\t\t", end=" ")
for i in r:
    print(i, end="\t\t\t\t")
print()
print("====== String Parsing =====")

for j in r:
    print("Terminal : ", j)
    for i in d.keys():
        if len(t[i][j]) != 0:
            print(i, "->", t[i][j])

    print()

string = "India won the championship"
parseString(string,t)
```

Input:

```
Enter the Production 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~a/an/the
===>
```

Output:

```
===== FIRST =====
first( S ) :  {'the', 'you', 'John', 'Steve', 'me', 'a', 'an', 'India', 'Australia', 'I'}
first( NP ) :  {'the', 'you', 'John', 'Steve', 'me', 'a', 'an', 'India', 'Australia', 'I'}
first( VP ) :  {'want', 'is', 'won', 'played'}
first( N ) :  {'championship', 'toss', 'ball'}
first( V ) :  {'want', 'is', 'won', 'played'}
first( P ) :  {'me', 'I', 'you'}
first( PN ) :  {'Steve', 'John', 'India', 'Australia'}
first( D ) :  {'a', 'an', 'the'}
===== FOLLOW =====
follow( S ) :  {'$'}
follow( NP ) :  {'want', '$', 'won', 'is', 'played'}
follow( VP ) :  {'$'}
follow( N ) :  {'want', '$', 'won', 'is', 'played'}
follow( V ) :  {'the', 'you', 'John', 'Steve', 'me', 'a', 'an', 'India', 'Australia', 'I'}
follow( P ) :  {'want', '$', 'won', 'is', 'played'}
follow( PN ) :  {'want', '$', 'won', 'is', 'played'}
follow( D ) :  {'championship', 'toss', 'ball'}
```

```
===== Parsing Table =====
                championship                        ball                        toss                        is
want                        won                         played                      me
I                           you                         India                       Australia
Steve                       John                        a                           an
the                         $
====== String Parsing =====
Terminal :  championship
N -> {'championship'}

Terminal :  ball
N -> {'ball'}

Terminal :  toss
N -> {'toss'}
```

```
Terminal :  is
VP -> {'V NP'}
V -> {'is'}

Terminal :  want
VP -> {'V NP'}
V -> {'want'}

Terminal :  won
VP -> {'V NP'}
V -> {'won'}

Terminal :  played
VP -> {'V NP'}
V -> {'played'}
```

```
Terminal :  me
S -> {'NP VP'}
NP -> {'P'}
P -> {'me'}

Terminal :  I
S -> {'NP VP'}
NP -> {'P'}
P -> {'I'}

Terminal :  you
S -> {'NP VP'}
NP -> {'P'}
P -> {'you'}
```

```
Terminal :  India
S -> {'NP VP'}
NP -> {'PN'}
PN -> {'India'}

Terminal :  Australia
S -> {'NP VP'}
NP -> {'PN'}
PN -> {'Australia'}

Terminal :  Steve
S -> {'NP VP'}
NP -> {'PN'}
PN -> {'Steve'}
```

```
Terminal :  John
S -> {'NP VP'}
NP -> {'PN'}
PN -> {'John'}

Terminal :  a
S -> {'NP VP'}
NP -> {'D N'}
D -> {'a'}

Terminal :  an
S -> {'NP VP'}
NP -> {'D N'}
D -> {'an'}
```

```
D -> {'an'}

Terminal :  the
S -> {'NP VP'}
NP -> {'D N'}
D -> {'the'}

Terminal :  $
```

```
Terminal :  $

Given String is :  India won the championship
Input                                                   Stack
['$', 'championship', 'the', 'won', 'India']                                    ['S', '$']
['NP', 'VP', '$']                                               ['NP', 'VP', '$']
['PN', 'VP', '$']                                               ['PN', 'VP', '$']
['India', 'VP', '$']                                            ['India', 'VP', '$']
['VP', '$']                                             ['VP', '$']
['V', 'NP', '$']                                                ['V', 'NP', '$']
['won', 'NP', '$']                                              ['won', 'NP', '$']
['NP', '$']                                             ['NP', '$']
['D', 'N', '$']                                                 ['D', 'N', '$']
['the', 'N', '$']                                               ['the', 'N', '$']
['N', '$']                                              ['N', '$']
['championship', '$']                                           ['championship', '$']
['$']                                   ['$']
***** String Accepted *****
[]                                      []
```

Virtual Lab Simulation:

1. Write your LL(1) grammar (empty string '' represents ε):

```
S ::= NP VP
NP ::= P
NP ::= PN
NP ::= D N
VP ::= V NP
N ::= championship
N ::= ball
N ::= toss
V ::= is
V ::= want
V ::= won
V ::= played
P ::= me
P ::= I
P ::= you
```

1. Write your LL(1) grammar (empty string '' represents ε):

```
N ::= toss
V ::= is
V ::= want
V ::= won
V ::= played
P ::= me
P ::= I
P ::= you
PN ::= India
PN ::= Australia
PN ::= Steve
PN ::= John
D ::= a
D ::= the
D ::= an
```

2. Nullable/First/Follow Table and Transition Table

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

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

3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ S

Remaining Input

India won the championship $

Rule

Partial Parse Tree

S

3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ VP NP

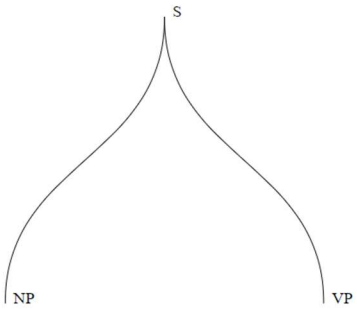
Remaining Input

India won the championship $

Rule

S ::= NP VP

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ VP PN

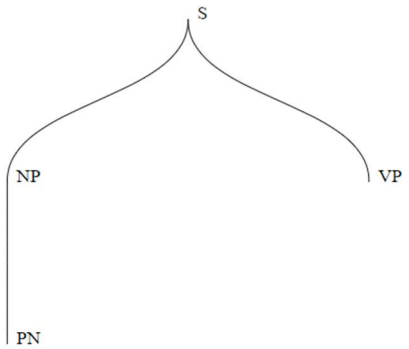
Remaining Input

India won the championship $

Rule

NP ::= PN

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ VP India

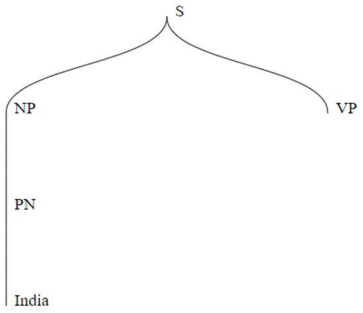
Remaining Input

India won the championship $

Rule

PN ::= India

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ VP

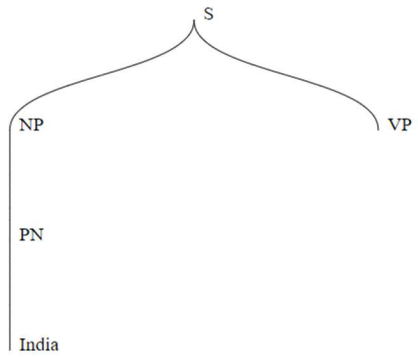
Remaining Input

won the championship $

Rule

Match India

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ NP V

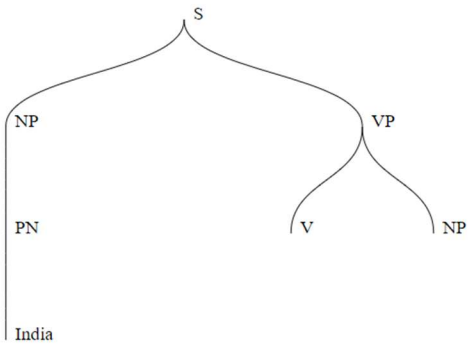
Remaining Input

won the championship $

Rule

VP ::= V NP

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ NP won

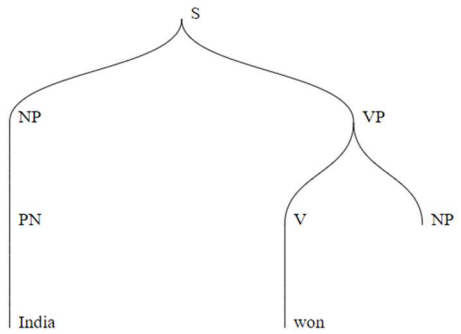
Remaining Input

won the championship $

Rule

V ::= won

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ NP

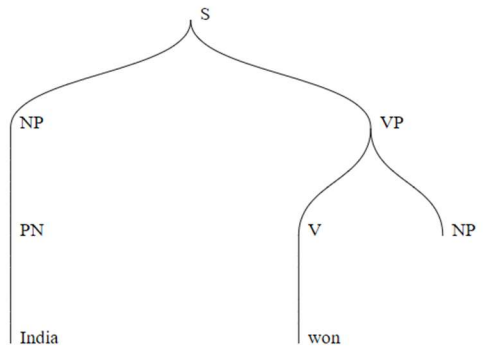
Remaining Input

the championship $

Rule

Match won

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ N D

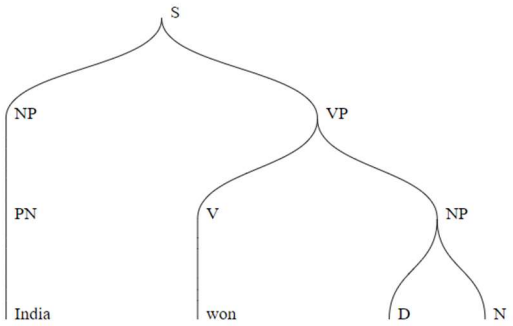
Remaining Input

the championship $

Rule

NP ::= D N

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ N the

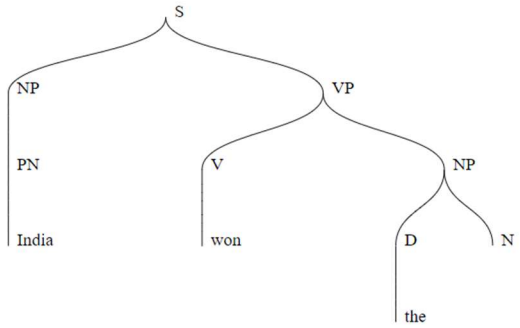
Remaining Input

the championship $

Rule

D ::= the

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ N

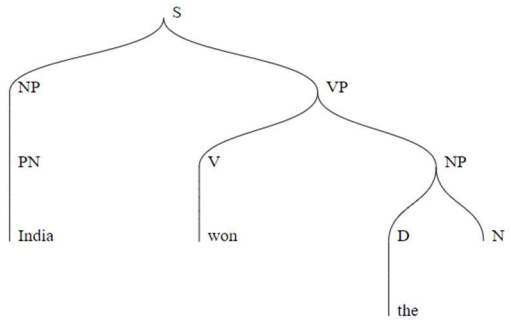
Remaining Input

championship $

Rule

Match the

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset | Step Forward

Stack

$ championship

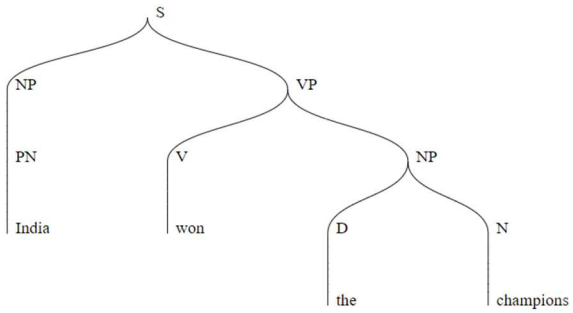
Remaining Input

championship $

Rule

N ::= championship

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset Step Forward

Stack

$

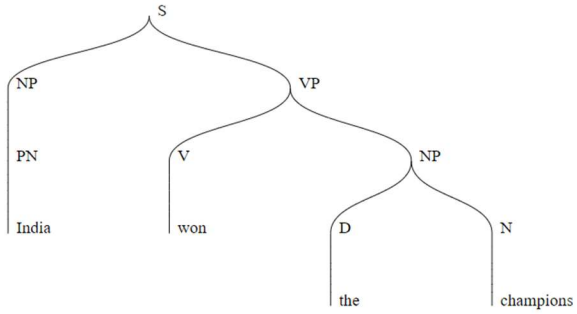
Remaining Input

$

Rule

Match championship

Partial Parse Tree



3. Parsing

Token stream separated by spaces: India won the championship

Start/Reset Step Forward

Stack

Remaining Input

Rule

Match $

Partial Parse Tree