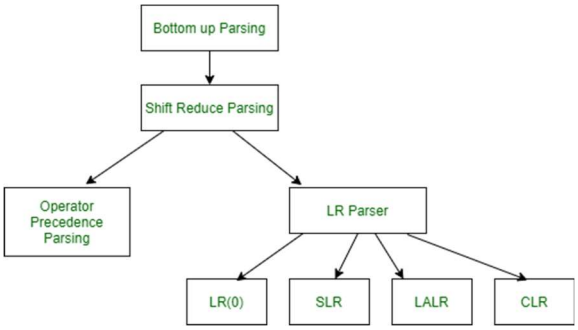
Name: Bhavesh Kewalramani
Roll No.: A-25

Practical No. 5

Theory

LR parser :

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.



Description of LR parser:

The term parser LR(k) parser, here the L refers to the left-to-right scanning, R refers to the rightmost derivation in reverse and k refers to the number of unconsumed "look ahead" input symbols that are used in making parser decisions. Typically, k is 1 and is often omitted. A context-free grammar is called LR (k) if the LR (k) parser exists for it. This first reduces the sequence of tokens to the left. But when we read from above, the derivation order first extends to non-terminal.

1. The stack is empty, and we are looking to reduce the rule by S'→S$.
2. Using a "." in the rule represents how many of the rules are already on the stack.
3. A dotted item, or simply, the item is a production rule with a dot indicating how much RHS has so far been recognized. Closing an item is used to see what production rules can be used to expand the current structure. It is calculated as follows:

Rules for LR parser :

The rules of LR parser as follows.

1. The first item from the given grammar rules adds itself as the first closed set.
2. If an object is present in the closure of the form A→ α. β. γ, where the next symbol after the symbol is non-terminal, add the symbol's production rules where the dot precedes the first item.
3. Repeat steps (B) and (C) for new items added under (B).

LR parser algorithm :

LR Parsing algorithm is the same for all the parser, but the parsing table is different for each parser. It consists following components as follows.

1. **Input Buffer–**
   It contains the given string, and it ends with a $ symbol.

2. **Stack–**
   The combination of state symbol and current input symbol is used to refer to the parsing table in order to take the parsing decisions.

Parsing Table :

Parsing table is divided into two parts- Action table and Go-To table. The **action table** gives a grammar rule to implement the given current state and current terminal in the input stream. There are four cases used in action table as follows.

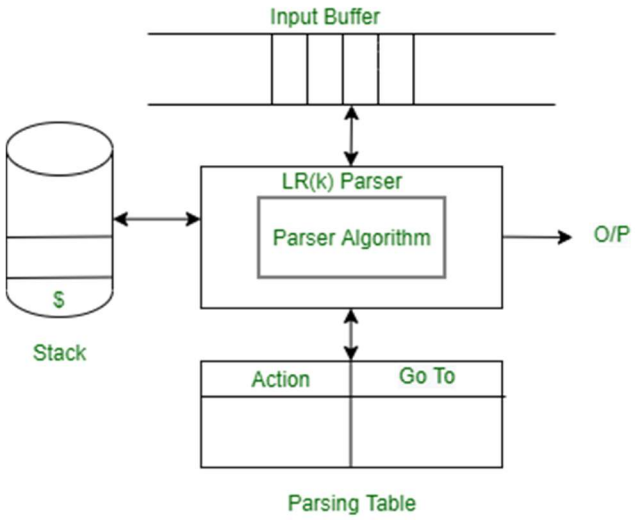
1. Shift Action- In shift action the present terminal is removed from the input stream and the state ***n*** is pushed onto the stack, and it becomes the new present state.
2. Reduce Action- The number ***m*** is written to the output stream.
3. The symbol ***m*** mentioned in the left-hand side of rule ***m*** says that state is removed from the stack.
4. The symbol ***m*** mentioned in the left-hand side of rule **m** says that a new state is looked up in the goto table and made the new current state by pushing it onto the stack.

```
An accept - the string is accepted

No action - a syntax error is reported
```

Note–
The **go-to table** indicates which state should proceed.

LR parser diagram :



Practicals

Aim:
Construct the SLR parser for the given grammar

Program:

```
def closure(I,nonterm):
    temp = I
    for prod in temp :
        index = prod[1].index('.')
        if(index<(len(prod[1])-1) and prod[1][index+1] in nonterm):
            for production in nonterm[prod[1][index+1]]:
                if([prod[1][index+1],str('.')+str(production)] not in temp):
                    temp.append([prod[1][index+1],str('.')+str(production)])
    return temp

state = []
I = []
def GoTo(start,nonterm,term):
    I.append(closure([[start+"",'.'+start+""]],nonterm))
    term += list(nonterm.keys())
    for ind in I:
        for grammar in term:
            if(grammar == ''):
                continue
            goto = False
            goto1 = False
            goto2 = False
```

```
        goto3 = False
        close = []
        for prod in ind:
            if(prod[1].index('.')<(len(prod[1])-1) and prod[1][prod[1].index('.')+1] is grammar):

close.append([prod[0],prod[1][:prod[1].index('.')]+grammar+'.'+prod[1][prod[1].index('.')+2:]])
        l = closure(close,nonterm)
        if(len(l) == 0):
            continue
        if(grammar in nonterm.keys()):
            goto1 = True
        else:
            goto3 = True
        if(l not in I):
            if(goto1):
                state.append([I.index(ind)+1,len(I)+1,grammar])
                goto = True
            elif(goto3):
                goto2 = True
                state.append([I.index(ind)+1,len(I)+1,grammar])
            I.append(l)

        else:
            if(goto1):
                goto = True
                state.append([I.index(ind)+1,I.index(l)+1,grammar])
            elif(goto3):
                goto2 = True
                state.append([I.index(ind)+1,I.index(l)+1,grammar])



terminals = []
nonTerminals = dict()
terminals = input("Enter the Terminals (|) : ").split("|")
n = int(input("Enter the number of Non-Terminals : "))

for i in range(n):
    ch = input("NonTerminals : ").strip()
    rules = input("Productions (|) : ").split("|")
    nonTerminals[ch] = rules

S = input("Start Symbol :  ")
terminals+=[]
print("Productions : ")
for i in nonTerminals.keys():
```

```
    print(i," -> ",end=' ')
    for j in nonTerminals[i]:
        print(j,end= ' | ')
    print()

GoTo(S,nonTerminals,terminals)
print("I States: ")
for count , i in enumerate(I):
    print(count+1 , i)

print("Transitions : ")
for count , i in enumerate(state):
    print(count+1, i)
```

Input:

```
Enter the Terminals (|) : +|*|(|)|id
Enter the number of Non-Terminals : 3
NonTerminals : E
Productions (|) : E+T|T
NonTerminals : T
Productions (|) : T*F|F
NonTerminals : F
Productions (|) : (E)|id
Start Symbol :  E
```

Output:

```
Start Symbol :  E
Productions :
E  ->  E+T | T |
T  ->  T*F | F |
F  ->  (E) | id |
I States:
1 [["E'", '.E'], ['E', '.E+T'], ['E', '.T'], ['T', '.T*F'], ['T', '.F'], ['F', '.(E)'], ['F', '.id']]
2 [['F', '(.E)'], ['E', '.E+T'], ['E', '.T'], ['T', '.T*F'], ['T', '.F'], ['F', '.(E)'], ['F', '.id']]
3 [["E'", 'E.'], ['E', 'E.+T']]
4 [['E', 'T.'], ['T', 'T.*F']]
5 [['T', 'F.']]
6 [['F', '(E.)'], ['E', 'E.+T']]
7 [['E', 'E+.T'], ['T', '.T*F'], ['T', '.F'], ['F', '.(E)'], ['F', '.id']]
8 [['T', 'T*.F'], ['F', '.(E)'], ['F', '.id']]
9 [['F', '(E).']]
10 [['E', 'E+T.'], ['T', 'T.*F']]
11 [['T', 'T*F.']]
```

```
Transitions :
1 [1, 2, '(']
2 [1, 3, 'E']
3 [1, 4, 'T']
4 [1, 5, 'F']
5 [2, 2, '(']
6 [2, 6, 'E']
7 [2, 4, 'T']
8 [2, 5, 'F']
9 [3, 7, '+']
10 [4, 8, '*']
11 [6, 7, '+']
12 [6, 9, ')']
13 [7, 2, '(']
14 [7, 10, 'T']
15 [7, 5, 'F']
16 [8, 2, '(']
17 [8, 11, 'F']
18 [10, 8, '*']
```

Input:

```
Enter the Terminals (|) : a|b
Enter the number of Non-Terminals : 2
NonTerminals : S
Productions (|) : AA
NonTerminals : A
Productions (|) : aA|b
Start Symbol :  S
```

Output:

```
Productions :
S  ->  AA |
A  ->  aA | b |
I States:
1 [["S'", '.S'], ['S', '.AA'], ['A', '.aA'], ['A', '.b']]
2 [['A', 'a.A'], ['A', '.aA'], ['A', '.b']]
3 [['A', 'b.']]
4 [["S'", 'S.']]
5 [['S', 'A.A'], ['A', '.aA'], ['A', '.b']]
6 [['A', 'aA.']]
7 [['S', 'AA.']]
Transitions :
1 [1, 2, 'a']
2 [1, 3, 'b']
3 [1, 4, 'S']
4 [1, 5, 'A']
5 [2, 2, 'a']
6 [2, 3, 'b']
7 [2, 6, 'A']
8 [5, 2, 'a']
9 [5, 3, 'b']
10 [5, 7, 'A']
```