GHARDA FOUNDATION'S
GHARDA INSTITUTE OF TECHNOLOGY, LAVEL
COMPUTER ENGINEERING DEPARTMENT
A/P: Lavel, Tal.Khed Dist. Ratnagiri

Evaluation Sheet

Class: TE-Computer

Semester: VI

Subject: **System Programming & Compiler Construction**

Experiment No: 5

Title of Experiment: Implementation of LL(1) Parser.

Sr. No.	Evaluation Criteria	Max Marks	Marks Obtained
1	Practical Performance	10	
2	Oral	5	
	Total	15	

Signature of Subject Teacher

Theory:

LL(1) Parser

LL(1) parser is a type of top-down parser that uses a deterministic finite automaton to parse an input string. It reads the input string from left to right and constructs the parse tree from left to right, applying the leftmost derivation at each step.

The "LL" in LL(1) stands for "Left-to-right, Leftmost derivation", which means that the parser starts parsing from the leftmost symbol of the input string and derives the leftmost non-terminal in each step until it reaches the start symbol of the grammar. This process is called a leftmost derivation.

The "1" in LL(1) stands for "1 token lookahead", which means that the parser only looks ahead one token in the input string to determine which production rule to apply at each step. This lookahead is necessary to resolve any potential conflicts that may arise due to the ambiguity in the grammar.

To construct an LL(1) parser, we need to generate a predictive parsing table that maps each non-terminal and lookahead symbol to the corresponding production rule to be applied. This parsing table is constructed based on the first and follow sets of the non-terminals in the grammar.

The "first" set of a non-terminal is the set of terminals that can appear as the first symbol in a derivation of that non-terminal. The "follow" set of a non-terminal is the set of terminals that can appear immediately after that non-terminal in any derivation of the grammar.

Using these sets, we can construct the predictive parsing table by iterating over each non-terminal and each terminal in the grammar and filling in the table entries accordingly. If there is a conflict in the table, then the grammar is not LL(1) parsable.

Once the parsing table is constructed, the LL(1) parser can parse the input string by maintaining a stack of symbols and a lookahead pointer. At each step, the parser checks the top of the stack and the lookahead symbol to determine which production rule to apply, and then pushes the corresponding symbols onto the stack. The parsing process continues until the input string is fully parsed and a valid parse tree is constructed.

Working of LL(1) Parser –

1. Grammar Analysis:

   The first step is to analyze the grammar and determine whether it is LL(1) parsable. This involves checking for the following properties:

   - The grammar is unambiguous.
   - The grammar is left-factored, i.e., there are no common prefixes in the production rules.
   - The grammar is free of left recursion.

2. First and Follow Sets:

   The next step is to compute the first and follow sets for each non-terminal in the grammar. The "first" set of a non-terminal is the set of terminals that can appear as the first symbol in a derivation of that non-terminal. The "follow" set of a non-terminal is the set of terminals that can appear immediately after that non-terminal in any derivation of the grammar.

3. Parsing Table Construction:

   Using the first and follow sets, we can construct the predictive parsing table, which maps each non-terminal and lookahead symbol to the corresponding production rule to be applied. Each entry in the table corresponds to a particular combination of non-terminal and lookahead symbol. If there is a conflict in the table, then the grammar is not LL(1) parsable.

4. Parsing of Input String:

   The final step is to parse the input string using the parsing table. The parser maintains a stack of symbols and a lookahead pointer, and it reads the input string from left to right. At each step, the parser checks the top of the stack and the lookahead symbol to determine which production rule to apply, and then pushes the corresponding symbols onto the stack. The parsing process continues until the input string is fully parsed, and a valid parse tree is constructed.

During the parsing process, the parser may encounter one of three situations:

- Predict: If the top of the stack is a non-terminal, and the lookahead symbol is in the first set of that non-terminal, the parser predicts which production rule to apply based on the parsing table.
- Match: If the top of the stack is a terminal, and it matches the lookahead symbol, the parser consumes the symbol and moves to the next one in the input string.
- Error: If there is no entry for the current combination of non-terminal and lookahead symbol in the parsing table, the parser reports an error.

Program Code –

```
def removeLeftRecursion(rulesDiction):
    store = {}

    for lhs in rulesDiction:
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]
        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
                betaRules.append(subrhs)
        if len(alphaRules) != 0:
            lhs_ = lhs + "'"
            while (lhs_ in rulesDiction.keys()) or (lhs_ in store.keys()):
                lhs_ += "'"
            for b in range(0, len(betaRules)):
                betaRules[b].append(lhs_)
            rulesDiction[lhs] = betaRules
```

```
            for a in range(0, len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(["#"])
            store[lhs_] = alphaRules
    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction
def LeftFactoring(rulesDiction):
    newDict = {}
    for lhs in rulesDiction:
        allrhs = rulesDiction[lhs]
        temp = dict()
        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)
        new_rule = []
        tempo_dict = {}
        for term_key in temp:
            allStartingWithTermKey = temp[term_key]
            if len(allStartingWithTermKey) > 1:
                lhs_ = lhs + "'"
                while (lhs_ in rulesDiction.keys()) or (lhs_ in
tempo_dict.keys()):
                    lhs_ += "'"
                new_rule.append([term_key, lhs_])
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
```

```
                tempo_dict[lhs_] = ex_rules
            else:
                new_rule.append(allStartingWithTermKey[0])
        newDict[lhs] = new_rule
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]
    return newDict



def first(rule):
    global rules, nonterm_userdef, term_userdef, diction, firsts

    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == "#":
            return "#"
    if len(rule) != 0:
        if rule[0] in list(diction.keys()):
            fres = []
            rhs_rules = diction[rule[0]]
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)

            if "#" not in fres:
```

```
                return fres
            else:
                newList = []
                fres.remove("#")
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList

                fres.append("#")
                return fres
def follow(nt):
    global start_symbol, rules, nonterm_userdef, term_userdef, diction,
firsts, follows
    solset = set()
    if nt == start_symbol:
        solset.add("$")

    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            if nt in subrule:
                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1 :]
```

```
                    if len(subrule) != 0:
                        res = first(subrule)
                        if "#" in res:
                            newList = []
                            res.remove("#")
                            ansNew = follow(curNT)
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        if nt != curNT:
                            res = follow(curNT)
                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
    return list(solset)
def computeAllFirsts():
    global rules, nonterm_userdef, term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
```

```
        rhs = k[1]
        multirhs = rhs.split("|")

        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs

    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")
    print(f"\nAfter elimination of left recursion:\n")

    diction = removeLeftRecursion(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")
    print("\nAfter left factoring:\n")
    diction = LeftFactoring(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")
    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)
```

```
        firsts[y] = t
    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
        print(f"first({key_list[index]}) " f"=> {firsts.get(gg)}")
        index += 1
def computeAllFollows():
    global start_symbol, rules, nonterm_userdef, term_userdef, diction,
firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            for g in sol:
                solset.add(g)
        follows[NT] = solset


    print("\nCalculated follows: ")
    key_list = list(follows.keys())
    index = 0
    for gg in follows:
        print(f"follow({key_list[index]})" f" => {follows[gg]}")
        index += 1
# create parse table
def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    print("\nFirsts and Follow Result table\n")
    # find space size
```

```
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2
    print(
        f"{{:<{10}}} "
        f"{{:<{mx_len_first + 5}}} "
        f"{{:<{mx_len_fol + 5}}}".format("Non-T", "FIRST", "FOLLOW")
    )
    for u in diction:
        print(
            f"{{:<{10}}} "
            f"{{:<{mx_len_first + 5}}} "
            f"{{:<{mx_len_fol    +    5}}}".format(u,    str(firsts[u]),
str(follows[u]))
        )
    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append("$")


    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append("")
```

```
        mat.append(row)


    grammar_is_LL = True


    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y)


            if "#" in res:
                if type(res) == str:
                    firstFollow = []
                    fol_op = follows[lhs]
                    if fol_op is str:
                        firstFollow.append(fol_op)
                    else:
                        for u in fol_op:
                            firstFollow.append(u)
                    res = firstFollow
                else:
                    res.remove("#")
                    res = list(res) + list(follows[lhs])
            ttemp = []
            if type(res) is str:
                ttemp.append(res)
                res = copy.deepcopy(ttemp)
            for c in res:
                xnt = ntlist.index(lhs)
                yt = terminals.index(c)
                if mat[xnt][yt] == "":
```

```
                    mat[xnt][yt] = mat[xnt][yt] + f"{lhs}->{' '.join(y)}"
                else:
                    if f"{lhs}->{y}" in mat[xnt][yt]:
                        continue
                    else:
                        grammar_is_LL = False
                        mat[xnt][yt]  =  mat[xnt][yt]  +  f",{lhs}->{'
'.join(y)}"


    print("\nGenerated parsing table:\n")
    frmt = "{:>12}" * len(terminals)
    print(frmt.format(*terminals))


    j = 0
    for y in mat:
        frmt1 = "{:>12}" * len(y)
        print(f"{ntlist[j]} {frmt1.format(*y)}")
        j += 1
    return (mat, grammar_is_LL, terminals)
def validateStringUsingStackBuffer(
    parsing_table,    grammarll1,    table_term_list,    input_string,
term_userdef, start_symbol
):
    print(f"\nValidate String => {input_string}\n")
    if grammarll1 == False:
        return f"\nInput String = " f'"{input_string}"\n' f"Grammar is not
LL(1)"
    stack = [start_symbol, "$"]
    buffer = []
    input_string = input_string.split()
    input_string.reverse()
```

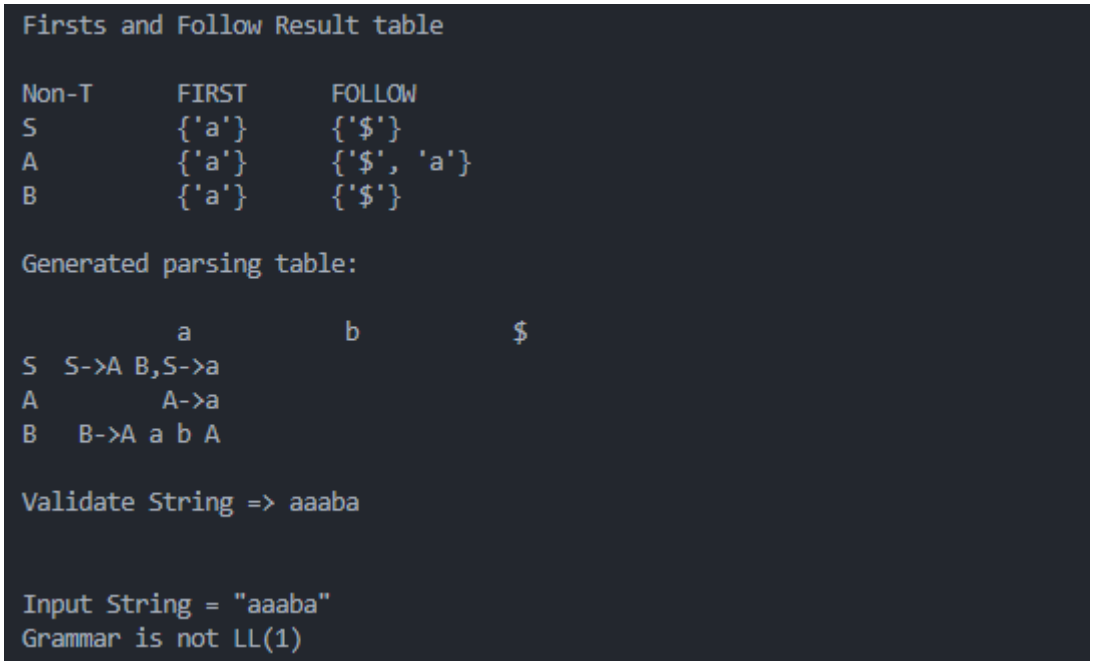
```
    buffer = ["$"] + input_string
    print("{:>20} {:>20} {:>20}".format("Buffer", "Stack", "Action"))
    while True:
        # end loop if all symbols matched
        if stack == ["$"] and buffer == ["$"]:
            print(
                "{:>20} {:>20} {:>20}".format(
                    " ".join(buffer), " ".join(stack), "Valid"
                )
            )
            return "\nValid String!"
        elif stack[0] not in term_userdef:
            # take font of buffer (y) and tos (x)
            x = list(diction.keys()).index(stack[0])
            y = table_term_list.index(buffer[-1])
            if parsing_table[x][y] != "":
                # format table entry received
                entry = parsing_table[x][y]
                print(
                    "{:>20} {:>20} {:>25}".format(
                        " ".join(buffer),
                        " ".join(stack),
                        f"T[{stack[0]}][{buffer[-1]}] = {entry}",
                    )
                )
                lhs_rhs = entry.split("->")
                lhs_rhs[1] = lhs_rhs[1].replace("#", "").strip()
                entryrhs = lhs_rhs[1].split()
                stack = entryrhs + stack[1:]
            else:
```

```
                return (
                    f"\nInvalid     String!     No     rule     at     "
f"Table[{stack[0]}][{buffer[-1]}]."
                )
        else:
            # stack top is Terminal
            if stack[0] == buffer[-1]:
                print(
                    "{:>20} {:>20} {:>20}".format(
                        "      ".join(buffer),      "      ".join(stack),
f"Matched:{stack[0]}"
                    )
                )
                buffer = buffer[:-1]
                stack = stack[1:]
            else:
                return "\nInvalid String! " "Unmatched terminal symbols"
rules = ["S -> A B | a", "A -> a", "B -> A a b A"]
nonterm_userdef = ["S", "A", "B"]
term_userdef = ["a", "b"]
sample_input_string = "aaaba"
diction = {}
firsts = {}
follows = {}

computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()

(parsing_table, result, tabTerm) = createParseTable()
```

```
if sample_input_string != None:
    validity = validateStringUsingStackBuffer(
        parsing_table, result, tabTerm, sample_input_string,
term_userdef, start_symbol
    )
    print(validity)
else:
    print("\nNo input String detected")
```

Output –

```
Rules:

S->[['A', 'B'], ['a']]
A->[['a']]
B->[['A', 'a', 'b', 'A']]

After elimination of left recursion:

S->[['A', 'B'], ['a']]
A->[['a']]
B->[['A', 'a', 'b', 'A']]

After left factoring:

S->[['A', 'B'], ['a']]
A->[['a']]
B->[['A', 'a', 'b', 'A']]

Calculated firsts:
first(S) => {'a'}
first(A) => {'a'}
first(B) => {'a'}

Calculated follows:
follow(S) => {'$'}
follow(A) => {'$', 'a'}
follow(B) => {'$'}
```



Conclusion –

Thus we studied the implementation of LL(1) parser.