Experiment -8. Write a program to Implement Shift Reduce parsing for a String.

Shift-reduce parsing is based on a bottom-up parsing technique, where the parser starts with the input tokens and aims to construct a parse tree by applying a set of grammar rules. The two main actions involved in shift-reduce parsing are “shift” and “reduce.”

Shift: During the shift operation, the parser reads the next input token and pushes it onto a stack. The parser maintains a buffer to hold the remaining input tokens. The shift operation moves the parser’s current position to the right in the input stream.

Reduce: The reduce operation applies a grammar rule to the tokens on top of the stack. If the tokens on the stack match the right-hand side of a grammar rule, they are replaced with the corresponding non-terminal symbol from the left-hand side of that rule. The reduce operation moves the parser’s current position to the left in the input stream.

Accept: If the stack contains the start symbol only and the input buffer is empty at the same time then that action is called accept.

Error: A situation in which the parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called an error action.

Shift-reduce parsing continues until it reaches the end of the input stream and constructs a valid parse tree. It employs a parsing table, often generated using techniques like the LR(1) or LALR(1) parsing algorithm, to determine the shift or reduce action to take at each step.

Example 1:

Consider the Grammar:
E->E+T | T
T->T*F | F
F->id

Code

```
#include <iostream>

#include <stack>

#include <vector>


using namespace std;


const vector<pair<string, string>> productions = {

  {"S", "E"},

  {"E", "E+T"},

  {"E", "T"},

  {"T", "T*F"},
```

```
    {"T", "F"},
    {"F", "(E)"},
    {"F", "id"}
};

void shift(stack<string> &p, string &i, int &j) {
    p.push(string(1, i[j++]));
}

void reduce(stack<string> &p) {
    for (const auto &r : productions) {
        string t, s;
        for (size_t i = 0; i < r.second.size(); ++i) {
            t = p.top() + t;
            p.pop();
        }
        if (t != r.second) {
            p.push(r.first);
            for (char c : t) {
                p.push(string(1, c));
            }
        }
    }
}

bool srParser(const string &i) {
    stack<string> p;
    int j = 0;
    while (j < i.size() || p.size() > 1) {
        if (j < i.size()) {
            shift(p, i, j);
```

```
        } else {
            reduce(p);
        }
    }
    return p.top() == "S";
}

int main() {
    string i;
    cout << "Enter the input string: ";
    cin >> i;

    if (srParser(i)) {
        cout << "Accepted" << endl;
    } else {
        cout << "Not Accepted" << endl;
    }

    return 0;
}
```

Output:

Example Input:

bash Copy code

```
id+id*id
```

Output:

python Copy code

```
Enter the input string: id+id*id
Accepted
```

Experiment -7. Write a program to find out the leading of the non-terminals in a grammar.

Leading of Non-terminals in a Grammar:

In a context-free grammar, the leading of a non-terminal refers to the set of terminals that can appear as the first symbols of strings derived from that non-terminal in a single step of a leftmost derivation. The leading is essential for predictive parsing, where the parser decides which production to use based on the next input symbol. To find the leading of a non-terminal, examine the production rules involving that non-terminal. If the production rule starts with the non-terminal itself or with other non-terminals leading to it, include the first symbols of those production rules in the leading set. Additionally, if the production rule starts with terminals, include them in the leading set. The leading information is crucial for constructing LL(1) parsing tables, ensuring unambiguous parsing with a top-down parser.

Code

```
#include <iostream>
#include <unordered_set>
#include <vector>
#include <algorithm>

using namespace std;

using Production = pair<string, string>;

unordered_set<char> findLeading(const vector<Production> &productions, char nonTerminal) {
    unordered_set<char> leading;
    for (const auto &production : productions) {
        if (production.first[0] == nonTerminal) {
            const char firstSymbol = production.second[0];
            if (isupper(firstSymbol)) {
                leading.insert(firstSymbol);
            } else {
                leading.insert(production.second.begin(), production.second.end());
            }
        }
    }
```

```
    return leading;
}

int main() {
    vector<Production> productions = {
        {"S", "aAb"},
        {"A", "c"},
        {"A", "d"},
        {"B", "e"},
        {"B", "f"}
    };

    cout << "Leading of A: ";
    for (char symbol : findLeading(productions, 'A')) {
        cout << symbol << " ";
    }
    cout << endl;

    cout << "Leading of B: ";
    for (char symbol : findLeading(productions, 'B')) {
        cout << symbol << " ";
    }
    cout << endl;

    return 0;
}
```

```
less                                                    Copy code

Leading of A: c d
Leading of B: e f
```

Experiment -6. Write a program to show all the operations of a stack.

Code

```
#include <iostream>
#include <stack>

using namespace std;

void pushOperation(stack<int> &s, int value) {
    s.push(value);
    cout << "Pushed " << value << " onto the stack." << endl;
}

void popOperation(stack<int> &s) {
    if (!s.empty()) {
        int topElement = s.top();
        s.pop();
        cout << "Popped " << topElement << " from the stack." << endl;
    } else {
        cout << "Stack is empty. Cannot perform pop operation." << endl;
    }
}

void displayOperation(stack<int> s) {
    cout << "Stack elements: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;
}
```

```
int main() {
    stack<int> myStack;

    // Push operation
    pushOperation(myStack, 10);
    pushOperation(myStack, 20);
    pushOperation(myStack, 30);

    // Display the stack
    displayOperation(myStack);

    // Pop operation
    popOperation(myStack);
    popOperation(myStack);
    popOperation(myStack);
    popOperation(myStack);  // Attempting pop when the stack is empty

    // Push more elements
    pushOperation(myStack, 40);
    pushOperation(myStack, 50);

    // Display the updated stack
    displayOperation(myStack);

    return 0;
}
```

```arduino
Pushed 10 onto the stack.
Pushed 20 onto the stack.
Pushed 30 onto the stack.
Stack elements: 30 20 10
Popped 30 from the stack.
Popped 20 from the stack.
Popped 10 from the stack.
Stack is empty. Cannot perform pop operation.
Pushed 40 onto the stack.
Pushed 50 onto the stack.
Stack elements: 50 40
```

Exp 5;-Write a program to perform Left Factoring on a Grammar.

theory ➖

Left Factoring in a Grammar:

1. Definition:

Left factoring is a technique used in grammar transformation to eliminate common prefixes in the right-hand sides of productions. It simplifies a grammar and aids in parser construction.

2. Common Prefixes:

A grammar has a common prefix if two or more productions of a non-terminal share the same initial sequence of symbols on their right-hand sides.

3. Issues with Common Prefixes:

Common prefixes can lead to ambiguity and difficulties in parsing, especially in top-down parsing techniques.

4. Left Factoring Steps:

a. Identify Common Prefixes:

- Identify common prefixes in the right-hand sides of productions for a given non-terminal.

b. Create New Non-terminal:

- For each set of productions sharing a common prefix, create a new non-terminal.

c. Split Productions:

- Replace the original productions with the new non-terminal, creating two sets of productions: one with the common prefix and one without.

Example:

Original Grammar:

A → αβ | αγ | δ

After Left Factoring:

A → αA'

A' → β | γ | ε

Example:

Original Grammar:

E → T + E | T

T → F * T | F

F → (E) | id

After Left Factoring:

E → T E'

E' → +E | ε

T → F T'

T' → *T | ε

F → (E) | id

CODE

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
struct ProductionRule {
   string nonTerminal;
   vector<string> productions;
};
void leftFactor(vector<ProductionRule>& grammar) {
   for (auto& rule : grammar) {
      string prefix = "";
      for (size_t i = 1; i < rule.productions.size(); ++i) {
         size_t j = 0;
         while (j < rule.productions[i - 1].size() && j < rule.productions[i].size() &&
               rule.productions[i - 1][j] == rule.productions[i][j]) {
            prefix += rule.productions[i - 1][j++];
         }
      }
      if (!prefix.empty()) {
         string newNonTerminal = rule.nonTerminal + "_factored";
         rule.productions = {prefix + newNonTerminal, prefix + "rest"};
         grammar.push_back({newNonTerminal, {rule.productions[0].substr(prefix.size())}});
         grammar.push_back({newNonTerminal + "_rest", {rule.productions[1].substr(prefix.size())}});
```

```
        }
    }
}

void printGrammar(const vector<ProductionRule>& grammar) {
    for (const auto& rule : grammar) {
        cout << rule.nonTerminal << " -> ";
        for (const auto& production : rule.productions)
            cout << production << " | ";
        cout << endl;
    }
}

int main() {
    vector<ProductionRule> grammar = {
        {"A", {"abc", "abd", "axyz"}},
        {"B", {"pqrs", "pqtu"}},
        // Add more production rules as needed
    };

    cout << "Original Grammar:\n";
    printGrammar(grammar);

    leftFactor(grammar);

    cout << "\nGrammar after Left Factoring:\n";
    printGrammar(grammar);

    return 0;
}
```

EXP 4:- Write a program to remove left Recursion from a Grammar.

theory :-

1. Definition:

Left recursion occurs in a production of a context-free grammar when a non-terminal A can directly derive a string beginning with itself. The presence of left recursion can cause issues in parsing and lead to infinite loops.

2. Types of Recursion:

Left Recursion: A → Aα

Right Recursion: A → αA

Indirect Recursion: A → Bβ, B → Aα

3. Issues with Left Recursion:

Left-recursive productions can lead to ambiguity and difficulties in top-down parsing. They hinder the construction of predictive parsers.

4. Left Factoring:

Before eliminating left recursion, it's common to perform left factoring. This technique reduces common prefixes in the right-hand side of productions.

5. Steps to Remove Left Recursion:

a. Identify Left-Recursive Productions:

- For a non-terminal A, if there exists a production A → Aα, it is left-recursive.

b. Create New Non-terminals:

- For each left-recursive non-terminal A, create a new non-terminal A'

(pronounced A prime).

c. Split Productions:

- Replace each left-recursive production A → Aα with two productions:

- A → βA'

- A' → αA' | ε (where ε is the empty string)

d. Adjust Other Productions:

- Adjust other productions involving A to use A' instead.

Example:

Original Grammar:

A → Aα | β

After Transformation:

A → βA'

A' → αA' | ε

Example:

Original Grammar:

E → E + T | T

T → T * F | F

F → (E) | id

After Removing Left Recursion:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

CODE

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

struct ProductionRule {
    string nonTerminal;
    vector<string> productions;
};

void removeLeftRecursion(vector<ProductionRule>& grammar) {
    for (size_t i = 0; i < grammar.size(); ++i) {
        for (size_t j = 0; j < i; ++j) {
            vector<string> newProductions;
            for (const auto& production : grammar[i].productions) {
                if (production[0] == grammar[j].nonTerminal[0]) {
                    for (const auto& alpha : grammar[j].productions) {
                        newProductions.push_back(alpha + production.substr(1));
                    }
                } else {
                    newProductions.push_back(production);
                }
            }
            grammar[i].productions = newProductions;

            vector<string> newNonTerminalProductions;
            for (const auto& production : grammar[j].productions) {
                newNonTerminalProductions.push_back(production + grammar[i].nonTerminal + "'");
            }
```

```
            grammar.push_back({grammar[i].nonTerminal + "'", newNonTerminalProductions});
        }

        vector<string> newProductions;
        vector<string> remainingProductions;
        for (const auto& production : grammar[i].productions) {
            if (production[0] == grammar[i].nonTerminal[0] && production.length() > 1) {
                remainingProductions.push_back(production.substr(1) + grammar[i].nonTerminal + "'");
            } else {
                newProductions.push_back(production + grammar[i].nonTerminal + "'");
            }
        }
        newProductions.push_back("epsilon");
        grammar[i].productions = newProductions;

        grammar.push_back({grammar[i].nonTerminal + "'", remainingProductions});
    }
}

void printGrammar(const vector<ProductionRule>& grammar) {
    for (const auto& rule : grammar) {
        cout << rule.nonTerminal << " -> ";
        for (const auto& production : rule.productions) {
            cout << production << " | ";
        }
        cout << endl;
    }
}

int main() {
    vector<ProductionRule> grammar = {
```

```
        {"E", {"E+T", "T"}},
        {"T", {"T*F", "F"}},
        {"F", {"(E)", "id"}},
        // Add more production rules as needed
    };

    cout << "Original Grammar:\n";
    printGrammar(grammar);

    removeLeftRecursion(grammar);

    cout << "\nGrammar after Removing Left Recursion:\n";
    printGrammar(grammar);

    return 0;
}
```

Experiment 2: Write a program to check whether a string belong to grammar

Theory:

Grammar Overview:

- Production Rules: The grammar specifies a set of production rules that define how valid strings are formed.
- Terminals and Non-terminals: Terminals are the actual symbols in the language (characters or tokens), while non-terminals represent groups of symbols.

String Checking Process:

Start with the Initial Symbol:

- Begin with the initial symbol of the grammar. This is often denoted by a non-terminal.

Apply Production Rules:

- Apply the production rules recursively, replacing non-terminals with the symbols defined by the rules.

Generate a String:

- Continue the process until you generate a string of terminals.

Compare with Input String:

- Compare the generated string with the input string to check if they match.

Example:

Consider a simple grammar:

Initial Symbol: S

Production Rules:

S → aS

S → Sb

S → ab

Now, let's check if the string "aab" belongs to this grammar:

Start:

Start with the initial symbol S.

Apply Production Rules:

S → aS (replace S with aS)

aS → aaS (replace S with aS again)

Generate String:

Continue applying rules: aaS → aab

Compare with Input String:

The generated string "aab" matches the input string "aab," so it belongs to the grammar.

String Rejection:

If, during the process, you reach a point where there's no rule to apply or the generated string doesn't match the input string, then the input string does not belong to the grammar.

Conclusion:

Checking whether a string belongs to a grammar involves systematically applying production rules to generate a string and comparing it with the input string. If they match, the string belongs to the grammar; otherwise, it doesn't. This process is fundamental to the theory of formal languages and automata.

Code:

```
#include <iostream>
#include <cctype>

bool isDigit(char c) {
    return std::isdigit(c);
}

bool isOperator(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/';
}

bool isExpression(const std::string& input) {
    for (char c : input) {
        if (!(isDigit(c) || isOperator(c) || std::isspace(c))) {
            return false;
        }
    }
    return true;
}
```

```
int main() {
    std::string input;
    std::cout << "Enter an arithmetic expression: ";
    std::getline(std::cin, input);

    if (isExpression(input)) {
        std::cout << "The string belongs to the grammar.\n";
    } else {
        std::cout << "The string does not belong to the grammar.\n";
    }

    return 0;
}
```

Experiment 3: write a program to check whether a string include keyword or not

Theory:

Keyword Checking Process:

Define Keywords:

Start by defining a list of keywords that you want to check for in the given string. Keywords are specific words or identifiers with a special meaning in a programming language.

Tokenization:

Tokenize the input string into individual units, such as words or symbols. This can be done using lexical analysis techniques.

Compare with Keywords:

Compare each token with the list of defined keywords. If a match is found, the string includes a keyword.

Handling Identifiers:

Be cautious when checking for keywords within identifiers. For example, in a programming language, if "int" is a keyword, you need to ensure that it's not mistakenly identified within a longer identifier like "interestingVariable."

Case Sensitivity:

Decide whether the keyword matching process is case-sensitive or case-insensitive. Some programming languages distinguish between uppercase and lowercase letters in keywords.

Example:

Consider a simple scenario in a programming language with keywords like "if," "else," and "while." Given the input string "if (x > 0) {", you would:

Define Keywords:

Keywords: "if," "else," "while."

Tokenization:

Tokenize the input string: ["if", "(", "x", ">", "0", ")", "{"]

Compare with Keywords:

Check each token against the list of keywords. In this case, "if" is found, indicating that the string includes a keyword.

String Rejection:

If, during the process, none of the tokens match the defined keywords, the string does not include any keywords. Alternatively, you might track the positions of the identified keywords within the string.

Conclusion:

Checking whether a string includes a keyword involves defining a set of keywords, tokenizing the string, and comparing the tokens with the keywords. This process is fundamental to lexical analysis, which is a crucial step in parsing and interpreting programming languages.

Code:

```
#include <iostream>
#include <algorithm>

int main() {
    std::string input, keyword;

    std::cout << "Enter a string: ";
```

```
    std::getline(std::cin, input);

    std::cout << "Enter the keyword to check: ";
    std::cin >> keyword;

    if (input.find(keyword) != std::string::npos) {
        std::cout << "String includes the keyword.\n";
    } else {
        std::cout << "String does not include the keyword.\n";
    }

    return 0;
}
```