

# Week 8 Assignment

## Problem

Implement a non-recursive Predictive Parser for the grammar.

$S \rightarrow aBa$

$B \rightarrow bB \mid \epsilon$

	A	b	\$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

## Program

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <set>
#include <stack>
#include <algorithm>
using namespace std;

class Grammar
{
private:
    int noOfProductions;
    int noOfTerminals;
    int noOfNonTerminals;
    map<char, vector<string>> grammar;
    set<char> nonTerminals;
    set<char> terminals;
    map<char, set<char>> first;
    map<string, set<char>> firstOfProduction;
    map<char, set<char>> follow;
    map<char, set<char>> oldFollow;
    vector<string> productions;
    set<char> calculateFirst(char ch)
    {
        set<char> firstSet;
        if (terminals.find(ch) != terminals.end())
        {
            firstSet.insert(ch);
        }
        else
```

```

    {
        for (string production : grammar[ch])
        {
            bool addEpsilon = true;
            for (char symbol : production)
            {
                set<char> symbolFirst = calculateFirst(symbol);
                firstSet.insert(symbolFirst.begin(), symbolFirst.end());

                if (symbolFirst.find('@') == symbolFirst.end())
                {
                    addEpsilon = false;
                    break;
                }
            }
            if (addEpsilon)
            {
                firstSet.insert('@');
            }
        }
    }
    return firstSet;
}

```

public:

```

Grammar() {}
Grammar(int noOfProuductions)
{
    this->noOfProductions = noOfProductions;
}
Grammar(int noOfTerminals, int noOfNonTerminals, int noOfProductions)
{
    this->noOfTerminals = noOfTerminals;
    this->noOfNonTerminals = noOfNonTerminals;
    this->noOfProductions = noOfProductions;
}
void setTerminals()
{
    char terminal;
    for (int i = 0; i < noOfTerminals; i++)
    {
        cin >> terminal;
        terminals.insert(terminal);
    }
}
void setNonTerminals()
{
    char nonTerminal;

```

```

        for (int i = 0; i < noOfNonTerminals; i++)
        {
            cin >> nonTerminal;
            nonTerminals.insert(nonTerminal);
        }
    }
    void setProductions()
    {
        char lhs;
        string rhs;
        string production;
        for (int i = 0; i < noOfProductions; i++)
        {
            cin >> production;
            productions.push_back(production);
            lhs = production[0];
            rhs = production.substr(3, production.length() - 3);
            // Ignores the -> symbol
            grammar[production[0]].push_back(rhs);
            nonTerminals.insert(production[0]);
        }
    }
    void setFirst()
    {
        set<char> firstSet;
        for (char nonTerminal : nonTerminals)
        {
            firstSet = calculateFirst(nonTerminal);
            first[nonTerminal].insert(firstSet.begin(), firstSet.end());
        }
    }
    void setFirstOfProductions()
    {
        for (string production : productions)
        {
            set<char> productionFirst;
            for (char symbol : production.substr(3, production.length() - 3))
            {
                set<char> symbolFirst = calculateFirst(symbol);
                productionFirst.insert(symbolFirst.begin(),
symbolFirst.end());
                if (symbolFirst.find('@') == symbolFirst.end())
                {
                    break; // Stop if epsilon is not in the first set of the
symbol
                }
            }
            firstOfProduction[production] = productionFirst;
        }
    }

```

```

    }
}
void setFollow()
{
    for (char nonTerminal : nonTerminals)
    {
        follow[nonTerminal] = {};
    }
    follow['$'].insert('$');
    bool changed = true;
    while (changed)
    {
        changed = false;

        for (char nonTerminal : nonTerminals)
        {
            for (auto it = grammar.begin(); it != grammar.end(); it++)
            {
                char leftHandSide = it->first;
                vector<string> productions = it->second;

                for (string production : productions)
                {
                    for (int i = 0; i < production.length(); i++)
                    {
                        if (production[i] == nonTerminal)
                        {
                            for (int j = i + 1; j < production.length();
j++)
                                {
                                    char symbol = production[j];
                                    if (terminals.find(symbol) !=
terminals.end())
                                    {
                                        follow[nonTerminal].insert(symbol);
                                        break;
                                    }
                                }
                        }
                        else
                        {
                            set<char> firstBeta =
calculateFirst(symbol);
                            if (firstBeta.find('@') ==
firstBeta.end())
                                {
                                    follow[nonTerminal].insert(firstBe
ta.begin(), firstBeta.end());
                                    break;
                                }
                        }
                    }
                }
            }
        }
    }
}

```

```

else
{
    firstBeta.erase('@');
    follow[nonTerminal].insert(firstBe
ta.begin(), firstBeta.end());

    if (j == production.length() - 1)
    {
        set<char> followA =
follow[leftHandSide];
        follow[nonTerminal].insert(fol
lowA.begin(), followA.end());
    }
}
}
}
if (i == production.length() - 1)
{
    set<char> followA = follow[leftHandSide];
    follow[nonTerminal].insert(followA.begin()
, followA.end());
}
}
}
}
}
for (char nonTerminal : nonTerminals)
{
    if (follow[nonTerminal] != oldFollow[nonTerminal])
    {
        changed = true;
        oldFollow[nonTerminal] = follow[nonTerminal];
    }
}
}
map<char, set<char>> getFirst()
{
    return first;
}
map<char, set<char>> getFollow()
{
    return follow;
}
map<string, set<char>> getFirstOfProductions()
{
    return firstOfProduction;
}

```

```

    }
    set<char> getTerminals()
    {
        return terminals;
    }
    set<char> getNonTerminals()
    {
        return nonTerminals;
    }
    vector<string> getProductions()
    {
        return productions;
    }
};

class LL1_Parser
{
private:
    Grammar CFG;
    map<pair<char, char>, string> parsing_table;

public:
    LL1_Parser() {}
    LL1_Parser(Grammar g)
    {
        this->CFG = g;
    }
    void createParsingTable()
    {
        set<char> nonTerminals = CFG.getNonTerminals();
        set<char> terminals = CFG.getTerminals();
        map<char, set<char>> follow = CFG.getFollow();
        map<string, set<char>> firstOfProductions =
CFG.getFirstOfProductions();
        for (auto production : CFG.getProductions())
        {
            set<char> firstOfPr = firstOfProductions[production];
            if (find(firstOfPr.begin(), firstOfPr.end(), '@') ==
firstOfPr.end())
            {
                for (char terminal : firstOfPr)
                {
                    parsing_table[make_pair(production[0], terminal)] =
production.substr(3);
                }
            }
            else
            {
                set<char> followOfNT = follow[production[0]];

```

```

        for (char terminal : followOfNT)
        {
            parsing_table[make_pair(production[0], terminal)] =
production.substr(3);
        }
    }
}

void display_parsing_table()
{
    for (auto entry : parsing_table)
    {
        cout << entry.first.first << "-" << entry.first.second << "-" <<
entry.second << endl;
    }
}

bool parse(string str)
{
    str += "$";
    stack<char> st;
    st.push('$');
    st.push('S');
    int ptr = 0;
    string production;
    set<char> nonTerminals = CFG.getNonTerminals();
    char symbol;
    do
    {
        symbol = st.top();
        if (find(nonTerminals.begin(), nonTerminals.end(), symbol) !=
nonTerminals.end())
        {
            if (parsing_table.count(make_pair(symbol, str[ptr])) <= 0)
            {
                return false;
            }
            else
            {
                st.pop();
                production = parsing_table[make_pair(symbol, str[ptr])];
                reverse(production.begin(), production.end());
                for (char ch : production)
                {
                    st.push(ch);
                }
            }
        }
        else

```

```

        {
            if (symbol == (char)str[ptr])
            {
                st.pop();
                ptr++;
            }
            else
            {
                return false;
            }
        }
        if (st.top() == '@')
        {
            st.pop();
        }
        // print_stack(st);
    } while (st.top() != '$');
    if ((char)str[ptr] == '$')
    {
        return true;
    }
    else
    {
        return false;
    }
}

void print_stack(stack<char> st)
{
    while (!st.empty())
    {
        cout << st.top();
        st.pop();
    }
    cout << endl;
}

};

int main()
{
    int noOfProductions, noOfTerminals, noOfNonTerminals;
    cout << "Enter the no of terminals: ";
    cin >> noOfTerminals;
    cout << "Enter the no of non terminals: ";
    cin >> noOfNonTerminals;
    cout << "Enter the no of productions: ";
    cin >> noOfProductions;
    Grammar g(noOfTerminals, noOfNonTerminals, noOfProductions);
    cout << "Enter the terminals" << endl;
    g.setTerminals();
}

```



```

cout << "Enter the non terminals" << endl;
g.setNonTerminals();
cout << "Enter the Productions" << endl;
g.setProductions();
g.setFirst();
g.setFollow();
g.setFirstOfProductions();
// map<char, set<char>> first = g.getFirst();
// map<char, set<char>> follow = g.getFollow();
// map<string, set<char>> firstOfProductions = g.getFirstOfProductions();
// for (auto pair : first)
// {
//     cout << "First(" << pair.first << ")"
//         << " = ";
//     for (auto terminal : pair.second)
//     {
//         cout << terminal << " ";
//     }
//     cout << endl;
// }
// for (auto pair : follow)
// {
//     cout << "Follow(" << pair.first << ")"
//         << " = ";
//     for (auto terminal : pair.second)
//     {
//         cout << terminal << " ";
//     }
//     cout << endl;
// }
// for (auto pair : firstOfProductions)
// {
//     cout << pair.first << "=";
//     for (auto terminal : pair.second)
//     {
//         cout << terminal << " ";
//     }
//     cout << endl;
// }
LL1_Parser l1(g);
// cout << "Parsing Table" << endl;
l1.createParsingTable();
// l1.display_parsing_table();
cout << "Enter the string: ";
string input;
cin >> input;
if (l1.parse(input))
{

```

```

        cout << "The can string can be generated usign the following grammar";
    }
    else
    {
        cout << "The strign can not be generated using the given grammar";
    }
    return 0;
}

```

## Input & Output:

```

Enter the no of terminals: 3
Enter the no of non terminals: 2
Enter the no of productions: 3
Enter the terminals
a
b
@
Enter the non terminals
S
B
Enter the Productions
S->aBa
B->bB
B->@
Enter the string: abbbbbbbbbbbbbbbbbbbbbbbbbbbba
The can string can be generated usign the following grammar

```