Shri Ramdeobaba College of Engineering and Management, Nagpur
Department of Computer Science and Engineering
Session: 2021-2022 [EVEN SEM]

Compiler Design Lab

Name : Janak Mandavgade

Sec : A

Roll no. : 43

Batch : A3

Subject : Compiler Design

PRACTICAL No. 3

Aim:

(A) Write a program to find FIRST for any grammar. All the following rules of FIRST must be implemented.

For a generalized grammar: A → αXY

FIRST (A) = FIRST (αXY)

= α if α is the terminal symbol (Rule-1)

= FIRST (α) if α is a non-terminal and FIRST (α) does not contain ε (Rule-2)

= FIRST (α) - ε ∪ FIRST (XY) if a is a non-terminal and FIRST (α) contains ε (Rule-3)

Input: Grammar rules from a file or from console entered by user.
Following inputs can be used:

Batch A1:
A SB | B
S a | Bc | □
B b | d

Batch A2:
S A | BC
A a | b
B p | □
C c

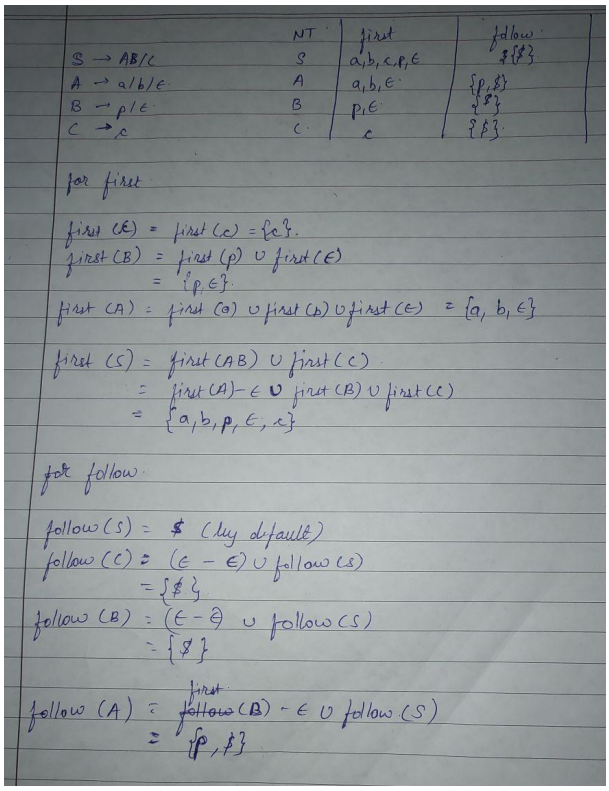
Batch A3:
S AB | C
A a | b | □
B p | □
C c

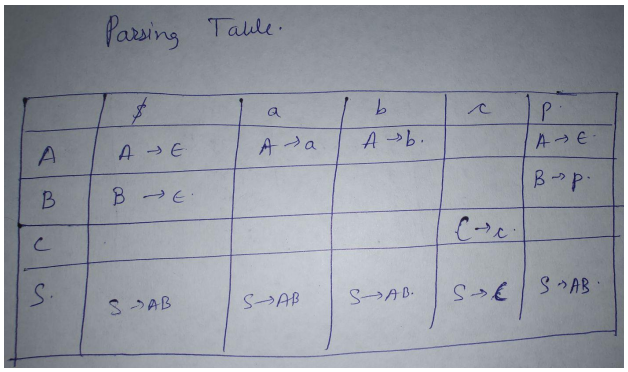
Batch A4:
S ABC | C
A a | bB | □
B p | □
C c

Implementation: FIRST rules
Output: FIRST information for each non-terminal

(B) Calculate Follow for the given grammar and Construct the LL (1) parsing table using the FIRST and FOLLOW.

Solved Numerical :





Code:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <bits/stdc++.h>

using namespace std;

void find_first(vector< pair<char, string> > gram,
        map< char, set<char> > &firsts,
        char non_term);
```

```
void find_follow(vector< pair<char, string> > gram,
	map< char, set<char> > &follows,
	map< char, set<char> > firsts,
	char non_term);


int main(int argc, char const *argv[])
{
	if(argc != 3) {
		cout<<"Arguments should be <grammar file> <input string>\n";
		return 1;
	}

	fstream grammar_file;
	grammar_file.open(argv[1], ios::in);
	if(grammar_file.fail()) {
		cout<<"Error in opening grammar file\n";
		return 2;
	}

	cout<<"Grammar parsed from grammar file: \n";
	vector< pair<char, string> > gram;
	int count = 0;
	while(!grammar_file.eof()) {
		char buffer[20];
		grammar_file.getline(buffer, 19);

		char lhs = buffer[0];
		string rhs = buffer+3;
		pair <char, string> prod (lhs, rhs);
		gram.push_back(prod);
		cout<<count++<<".                    "<<gram.back().first<<"          ->
"<<gram.back().second<<"\n";
	}
	cout<<"\n";

	set<char> non_terms;
	for(auto i = gram.begin(); i != gram.end(); ++i) {
		non_terms.insert(i->first);
	}
	cout<<"The non terminals in the grammar are: ";
	for(auto i = non_terms.begin(); i != non_terms.end(); ++i) {
		cout<<*i<<" ";
	}
	cout<<"\n";
	set<char> terms;
	for(auto i = gram.begin(); i != gram.end(); ++i) {
		for(auto ch = i->second.begin(); ch != i->second.end(); ++ch) {
			if(!isupper(*ch)) {
				terms.insert(*ch);
			}
```

```
                }
        }
        terms.erase('e');
        terms.insert('$');
        cout<<"The terminals in the grammar are: ";
        for(auto i = terms.begin(); i != terms.end(); ++i) {
                cout<<*i<<" ";
        }
        cout<<"\n\n";

        char start_sym = gram.begin()->first;


        map< char, set<char> > firsts;
        for(auto non_term = non_terms.begin(); non_term != non_terms.end();
++non_term) {
                if(firsts[*non_term].empty()){
                        find_first(gram, firsts, *non_term);
                }
        }

        cout<<"Firsts list: \n";
        for(auto it = firsts.begin(); it != firsts.end(); ++it) {
                cout<<it->first<<" : ";
                for(auto firsts_it = it->second.begin(); firsts_it != it->second.end();
++firsts_it) {
                        cout<<*firsts_it<<" ";
                }
                cout<<"\n";
        }
        cout<<"\n";


        map< char, set<char> > follows;
        char start_var = gram.begin()->first;
        follows[start_var].insert('$');
        find_follow(gram, follows, firsts, start_var);
        for(auto it = non_terms.begin(); it != non_terms.end(); ++it) {
                if(follows[*it].empty()) {
                        find_follow(gram, follows, firsts, *it);
                }
        }

        cout<<"Follows list: \n";
        for(auto it = follows.begin(); it != follows.end(); ++it) {
                cout<<it->first<<" : ";
                for(auto follows_it = it->second.begin(); follows_it != it->second.end();
++follows_it) {
                        cout<<*follows_it<<" ";
                }
                cout<<"\n";
```

```
	}
	cout<<"\n";


	int parse_table[non_terms.size()][terms.size()];
	fill(&parse_table[0][0], &parse_table[0][0] +
sizeof(parse_table)/sizeof(parse_table[0][0]), -1);
	for(auto prod = gram.begin(); prod != gram.end(); ++prod) {
		string rhs = prod->second;

		set<char> next_list;
		bool finished = false;
		for(auto ch = rhs.begin(); ch != rhs.end(); ++ch) {
			if(!isupper(*ch)) {
				if(*ch != 'e') {
					next_list.insert(*ch);
					finished = true;
					break;
				}
				continue;
			}

			set<char> firsts_copy(firsts[*ch].begin(), firsts[*ch].end());
			if(firsts_copy.find('e') == firsts_copy.end()) {
				next_list.insert(firsts_copy.begin(), firsts_copy.end());
				finished = true;
				break;
			}
			firsts_copy.erase('e');
			next_list.insert(firsts_copy.begin(), firsts_copy.end());
		}
		if(!finished) {
			next_list.insert(follows[prod->first].begin(),
follows[prod->first].end());
		}


		for(auto ch = next_list.begin(); ch != next_list.end(); ++ch) {
			int row = distance(non_terms.begin(), non_terms.find(prod->first));
			int col = distance(terms.begin(), terms.find(*ch));
			int prod_num = distance(gram.begin(), prod);
			if(parse_table[row][col] != -1) {
				cout<<"Collision at ["<<row<<"]["<<col<<"] for production
"<<prod_num<<"\n";
				continue;
			}
			parse_table[row][col] = prod_num;
		}

	}
	cout<<"Parsing Table: \n";
```

```
cout<<"   ";
for(auto i = terms.begin(); i != terms.end(); ++i) {
	cout<<*i<<" ";
}
cout<<"\n";
for(auto row = non_terms.begin(); row != non_terms.end(); ++row) {
	cout<<*row<<"  ";
	for(int col = 0; col < terms.size(); ++col) {
		int row_num = distance(non_terms.begin(), row);
		if(parse_table[row_num][col] == -1) {
			cout<<"- ";
			continue;
		}
		cout<<parse_table[row_num][col]<<" ";
	}
	cout<<"\n";
}
cout<<"\n";


string input_string(argv[2]);
input_string.push_back('$');
stack<char> st;
st.push('$');
st.push('S');

for(auto ch = input_string.begin(); ch != input_string.end(); ++ch) {
	if(terms.find(*ch) == terms.end()) {
		cout<<"Input string is invalid\n";
		return 2;
	}
}

bool accepted = true;
while(!st.empty() && !input_string.empty()) {

	if(input_string[0] == st.top()) {
		st.pop();
		input_string.erase(0, 1);
	}
	else if(!isupper(st.top())) {
		cout<<"Unmatched terminal found\n";
		accepted = false;
		break;
	}
	else {
		char stack_top = st.top();
		int row = distance(non_terms.begin(), non_terms.find(stack_top));
		int col = distance(terms.begin(), terms.find(input_string[0]));
		int prod_num = parse_table[row][col];
```

```
                        if(prod_num == -1) {
                                cout<<"No production found in parse table\n";
                                accepted = false;
                                break;
                        }

                        st.pop();
                        string rhs = gram[prod_num].second;
                        if(rhs[0] == 'e') {
                                continue;
                        }
                        for(auto ch = rhs.rbegin(); ch != rhs.rend(); ++ch) {
                                st.push(*ch);
                        }
                }
        }

        if(accepted) {
                cout<<"Input string is accepted\n";
        }
        else {
                cout<<"Input string is rejected\n";
        }

        return 0;
}

void find_first(vector< pair<char, string> > gram,
        map< char, set<char> > &firsts,
        char non_term) {

        for(auto it = gram.begin(); it != gram.end(); ++it) {
                if(it->first != non_term) {
                        continue;
                }

                string rhs = it->second;
                for(auto ch = rhs.begin(); ch != rhs.end(); ++ch) {
                        if(!isupper(*ch)) {
                                firsts[non_term].insert(*ch);
                                break;
                        }
                        else {
                                if(firsts[*ch].empty()) {
                                        find_first(gram, firsts, *ch);
                                }
                                if(firsts[*ch].find('e') == firsts[*ch].end()) {
                                        firsts[non_term].insert(firsts[*ch].begin(),
firsts[*ch].end());
                                        break;
                                }
```

```
                                        set<char> firsts_copy(firsts[*ch].begin(), firsts[*ch].end());
                                        if(ch + 1 != rhs.end()) {
                                                firsts_copy.erase('e');
                                        }
                                        firsts[non_term].insert(firsts_copy.begin(),
firsts_copy.end());
                                }
                        }

                }
}

void find_follow(vector< pair<char, string> > gram,
                map< char, set<char> > &follows,
                map< char, set<char> > firsts,
                char non_term) {

                // cout<<"Finding follow of "<<non_term<<"\n";

                for(auto it = gram.begin(); it != gram.end(); ++it) {
                        bool finished = true;
                        auto ch = it->second.begin();
                        for(;ch != it->second.end() ; ++ch) {
                                if(*ch == non_term) {
                                        finished = false;
                                        break;
                                }
                        }
                        ++ch;

                        for(;ch != it->second.end() && !finished; ++ch) {
                                if(!isupper(*ch)) {
                                        follows[non_term].insert(*ch);
                                        finished = true;
                                        break;
                                }

                                set<char> firsts_copy(firsts[*ch]);
                                if(firsts_copy.find('e') == firsts_copy.end()) {
                                        follows[non_term].insert(firsts_copy.begin(),
firsts_copy.end());
                                        finished = true;
                                        break;
                                }

                                firsts_copy.erase('e');
                                follows[non_term].insert(firsts_copy.begin(), firsts_copy.end());

                        }
```

```
                    if(ch == it->second.end() && !finished) {
                        if(follows[it->first].empty()) {
                            find_follow(gram, follows, firsts, it->first);
                        }
                        follows[non_term].insert(follows[it->first].begin(),
follows[it->first].end());
                    }

                }

        }
```

Output:

```
Grammar parsed from grammar file:
0.  S -> AB
1.  S -> C
2.  A -> a
3.  A -> b
4.  A -> e
5.  B -> p
6.  B -> e
7.  C -> c

The non terminals in the grammar are: A B C S
The terminals in the grammar are: $ a b c p

Firsts list:
A : a b e
B : e p
C : c
S : a b c e p

Follows list:
A : $ p
B : $
C : $
S : $
```

```
Grammar parsed from grammar file:
0.  S -> AB
1.  S -> C
2.  A -> a
3.  A -> b
4.  A -> e
5.  B -> p
6.  B -> e
7.  C -> c

The non terminals in the grammar are: A B C S
The terminals in the grammar are: $ a b c p

Firsts list:
A : a b e
B : e p
C : c
S : a b c e p

Follows list:
A : $ p
B : $
C : $
S : $

Parsing Table:
   $ a b c p
A  4 2 3 - 4
B  6 - - - 5
C  - - - 7 -
S  0 0 0 1 0
```