Practical: 11

Aim: Write a program which take a grammar as input and check that given grammar is LL(1) or not.

Program:

```
/*
    author: mr_bhishm
    created: 10-10-2020 19:02:19
    "Make it work, make it right, make it fast."
                                                    – Kent Beck
*/

#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include <map>
#include <stack>

using namespace std;


bool NotLL1 = false;
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;
      }
      // Arguments check
      // cout<<argv[1]<<argv[2];


      // Parsing the grammar file
      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";
    cout<<"[!] Here \'e\' for epsilon, $ used as end of string and any other symbol
treated as terminals."<<endl;
    cout<<"[!] Here Capital letters used for Not terminal/Variable."<<endl;
    cout<<"[!] If find any mistake according to this then exit and modify grammar and
rerun it."<<endl;
    cout<<endl;
	  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";


	  // Gather all non terminals
	  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";
	  // Gather all terminals
	  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);
		      }
		}
	  }
	  // Remove epsilon and add end character $
	  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";


      // Start symbol is first non terminal production in grammar
      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;
      // Find follow of start variable first
      char start_var = gram.begin()->first;
      follows[start_var].insert('$');
      find_follow(gram, follows, firsts, start_var);
      // Find follows for rest of variables
      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 the whole rhs can be skipped through epsilon or reaching the end
		// Add follow to next list
		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";
			    NotLL1 = true;
				continue;
			}
```

```
                parse_table[row][col] = prod_num;
            }

      }
      // Print parse table

      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";

    if(!NotLL1){
        cout<<"[+] Grammar is LL1..."<<endl;
        string input_string(argv[2]);
        string input_string_2 = input_string;
        input_string.push_back('$');
        stack<char> st;
        st.push('$');
        st.push('S');

        // Check if input string is valid
        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;
            }
        }

        // cout<<"Processing input string\n";
        bool accepted = true;
        while(!st.empty() && !input_string.empty()) {
            // If stack top same as input string char remove it

            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: \""<<input_string_2<<"\" is Accepted\n";
        }
        else {
            cout<<"[+] Input string is Rejected\n";
        }
    }else{
        cout<<"[!] Conflict in parse table entries"<<endl;
        cout<<"[+] Grammar is not LL1..."<<endl;
    }


      return 0;
}

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

```
	// cout<<"Finding firsts of "<<non_term<<"\n";

	for(auto it = gram.begin(); it != gram.end(); ++it) {
		// Find productions of the non terminal
		if(it->first != non_term) {
			continue;
		}

		// cout<<"Processing production "<<it->first<<"->"<<it->second<<"\n";

		string rhs = it->second;
		// Loop till a non terminal or no epsilon variable found
		for(auto ch = rhs.begin(); ch != rhs.end(); ++ch) {
			// If first char in production a non term, add it to firsts list
			if(!isupper(*ch)) {
				firsts[non_term].insert(*ch);
				break;
			}
			else {
				// If char in prod is non terminal and whose firsts has no yet
been found out
				// Find first for that non terminal
				if(firsts[*ch].empty()) {
					find_first(gram, firsts, *ch);
				}
				// If variable doesn't have epsilon, stop loop
				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());

				// Remove epsilon from firsts if not the last variable
				if(ch + 1 != rhs.end()) {
					firsts_copy.erase('e');
				}

				// Append firsts of that variable
				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) {

            // finished is true when finding follow from this production is complete
            bool finished = true;
            auto ch = it->second.begin();

            // Skip variables till reqd non terminal
            for(;ch != it->second.end() ; ++ch) {
                  if(*ch == non_term) {
                        finished = false;
                        break;
                  }
            }
            ++ch;

            for(;ch != it->second.end() && !finished; ++ch) {
                  // If non terminal, just append to follow
                  if(!isupper(*ch)) {
                        follows[non_term].insert(*ch);
                        finished = true;
                        break;
                  }

                  set<char> firsts_copy(firsts[*ch]);
                  // If char's firsts doesnt have epsilon follow search is over
                  if(firsts_copy.find('e') == firsts_copy.end()) {
                        follows[non_term].insert(firsts_copy.begin(),
firsts_copy.end());
                        finished = true;
                        break;
                  }
                  // Else next char has to be checked after appending firsts to follow
                  firsts_copy.erase('e');
                  follows[non_term].insert(firsts_copy.begin(), firsts_copy.end());

            }


            // If end of production, follow same as follow of variable
            if(ch == it->second.end() && !finished) {
                  // Find follow if it doesn't have
                  if(follows[it->first].empty()) {
```

```
                        find_follow(gram, follows, firsts, it->first);
                    }
                    follows[non_term].insert(follows[it->first].begin(), follows[it-
>first].end());
                }

            }

}

/*
0). input.txt

    S->(L)
    S->a
    L->S
    L->L,S

    Output: Not LL(1)

1). input1.txt

    S->F
    S->(S+F)
    F->a

    Output: LL(1)

2). input2.txt

    S->AB
    A->a
    A->e
    B->b
    B->e

    Output: LL(1)
*/
```

Output:

```
(base) PS D:\DLP_lab\Practical_11> g++ .\LL1.cpp
(base) PS D:\DLP_lab\Practical_11> .\a.exe .\input.txt a
[+] Grammar parsed from grammar file:
[!] Here 'e' for epsilon, $ used as end of string and any other symbol treated as terminals.
[!] Here Capital letters used for Not terminal/Variable.
[!] If find any mistake according to this then exit and modify grammar and rerun it.

[0].  S -> (L)
[1].  S -> a
[2].  L -> S
[3].  L -> L,S

[+] The non terminals in the grammar are: L S
[+] The terminals in the grammar are: $ ( ) , a

[+] Firsts list:
L : ( a
S : ( a

[+] Follows list:
L : ) ,
S : $ ) ,

[!] Collision at [0][1] for production 3
[!] Collision at [0][4] for production 3
[+] Parsing Table:
   $ ( ) , a
L  - 2 - - 2
S  - 0 - - 1

[!] Conflict in parse table entries
[+] Grammar is not LL1...
```

```
(base) PS D:\DLP_lab\Practical_11> .\a.exe .\input1.txt a
[+] Grammar parsed from grammar file:
[!] Here 'e' for epsilon, $ used as end of string and any other symbol treated as terminals.
[!] Here Capital letters used for Not terminal/Variable.
[!] If find any mistake according to this then exit and modify grammar and rerun it.

[0].  S -> F
[1].  S -> (S+F)
[2].  F -> a

[+] The non terminals in the grammar are: F S
[+] The terminals in the grammar are: $ ( ) + a

[+] Firsts list:
F : a
S : ( a

[+] Follows list:
F : $ ) +
S : $ +

[+] Parsing Table:
   $ ( ) + a
F  - - - - 2
S  - 1 - - 0

[+] Grammar is LL1...
[+] Input string: "a" is Accepted
```

Conclusion: From this practical I have learnt about how to generate parse table for given grammar and decide whether given grammar is LL1 or not.