4 - First & Follow Sets

Tuesday, 5 March 2024 1:19 AM

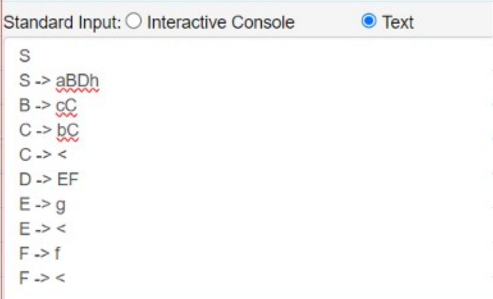
AIM: Compute First and Follow for a production and construct a predictive parser table for it

Procedure:

1. input start symbol and gramma from user and check that gramer is correctly formated.
2. check that all non-terminals are defined by the grammer rules.
3. calculate fist sets by iteratiy through each production rule and addiy it to the set if it is a terminal or epsilon or going to the production rule of the non-terminal to find its first set symbol.
4. Calculate follow sets by initializiy the start symbol's follow as $ in the set.
   - if a non-terminal is at the end of a production rule, for each production rule
     add it to the follow of the left hand side non-terminal.
5. print the first and follow sets.
6. Construct the parsing table entries for each production in the grammer. by parsiy the first and follow values.
7. use the constructed parsiy table to parse input strigs usiy a stack.

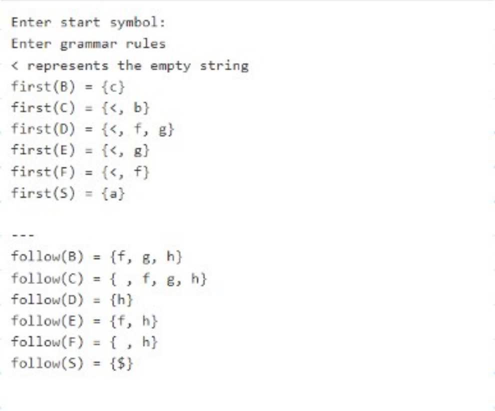
SAMPLE INPUT

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \epsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \epsilon$
$F \rightarrow f \mid \epsilon$



```
S
S -> aBDh
B -> cC
C -> bC
C -> <
D -> EF
E -> g
E -> <
F -> f
F -> <
```

OUTPUT



```
Enter start symbol:
Enter grammar rules
< represents the empty string
first(B) = {c}
first(C) = {<, b}
first(D) = {<, f, g}
first(E) = {<, g}
first(F) = {<, f}
first(S) = {a}

---
follow(B) = {f, g, h}
follow(C) = { , f, g, h}
follow(D) = {h}
follow(E) = {f, h}
follow(F) = { , h}
follow(S) = {$}
```

CODE:

```
#include <iostream>
#include <map>
#include <set>
#include <string>
```

```
#include <cctype>
#include <vector>
int read_rule(std::map<char, std::vector<std::string> > &rules) {
    char t;
    std::string checkarrow;
    std::string expansion;
    const std::string arrow ("->");
    std::cin >> t; // read terminal
    if (!isupper(t)) {
        return 1;
    }
    std::cin >> checkarrow;
    if (checkarrow.compare(arrow)) {
        return 2;
    }
    std::cin >> expansion;
    rules[t].push_back(expansion);
    return 0;
}
int is_invalid(std::map<char, std::vector<std::string> > &rules) {
    std::map<char, std::vector<std::string > >::iterator i;
    for (i = rules.begin(); i != rules.end(); i++) {
        int j;
        for (j = 0; j < (i->second).size(); j++) {
            std::string st = i->second[j];
            int k;
            if (st[0] == '<' && st.length() != 1) {
                return -1;
            }
            for (k = 0; k < st.length(); k++) {
                char token = st[k];
                if (isupper(token) && rules.find(token) == rules.end()) {
                    // nonterminal not defined in grammar
                    return (int)token;
                }
            }
        }
    }
    return 0;
}
int calculate_total_size(std::map<char, std::set<char> > &first) {
    int total = 0;
    std::map<char, std::set<char> >::iterator x;
    for (x = first.begin(); x != first.end(); x++) {
        total += (x->second).size();
    }
    return total;
}

void print_sets(std::map<char, std::set<char> > &first, std::string str) {
    std::map<char, std::set<char> >::iterator x;
    std::set<char>::iterator y;
    int i;
    for (x = first.begin(); x != first.end(); x++) {
        std::cout << str << "(" << x->first << ") = {";
        for (i = 0, y = (x->second).begin(); y != (x->second).end(); y++, i++) {
            if (i != 0) {
                std::cout << ", ";
            }
            std::cout << *y;
        }
        std::cout << "}" << std::endl;
    }
}

void calculate_first(std::map<char, std::set<char> > &first, std::map<char,
std::vector<std::string> > &rules) {
    int size_before_iteration, size_after_iteration;
    std::map<char, std::vector<std::string > >::iterator i;
    // 1: Initialize every first(T) = {}
    // This is done by default

    size_before_iteration = 0;

    while (true) {
    // 2: Add Fi(w) for every A -> w
        for (i = rules.begin(); i != rules.end(); i++) {
            int j;
            char terminal = i->first;
            for (j = 0; j < (i->second).size(); j++) {
                // For every rule
                std::string w = i->second[j];
                // A -> <
                // A -> aw
                if (w[0] == '<' || !isupper(w[0])) {
                    first[terminal].insert(w[0]);
                }
                // A -> Aw
                else if (isupper(w[0])) {
                    if (first[w[0]].find('<') == first[w[0]].end()) {
                        // epsilon not present in Fi(A)
                        first[terminal].insert(first[w[0]].begin(),
first[w[0]].end());
                    }
                    else {
                        // epsilon present in Fi(A)
                        std::set<char> first_w_A = first[w[0]];
                        first_w_A.erase('<');
                        if (w.length() > 1) {
                            // A -> Xw' (w = Xw')
                            if (isupper(w[1])) {
                                first_w_A.insert(first[w[1]].begin(),
first[w[1]].end());
                            }
                            else {
                                first_w_A.insert(w[1]);
                            }
                        }
                        first[terminal].insert(first_w_A.begin(),
first_w_A.end());
                    }
                }
            }
        }
        size_after_iteration = calculate_total_size(first);
        if (size_before_iteration == size_after_iteration) {
            // no change
            break;
        }
        else {
            // new "size before iteration"
            size_before_iteration = size_after_iteration;
        }
        //std::cout << "---" << std::endl;
        //print_sets(first, "first");
        //std::cout << "---" << std::endl;
        //std::cout << size_after_iteration << std::endl;
    }
    print_sets(first, "first");

}
// string is reductible to empty if it consists completely of nonterminals which
have epsilon in their first set
bool is_reductible_to_empty(std::string &str, std::map<char, std::set<char> >
&first) {
    int i;
    bool all_epsilon = true;
    for (i = 0; i < str.length(); i++) {
        if (isupper(str[i]) && (first[str[i]].find('<') != first[str[i]].end()))
{
            // each token is a terminal, and it contains epsilon in its follow
set
            continue;
        }
        else {
            //std::cout << "[LOG] " << str << " is not reductible to epsilon"
<< std::endl;
            return false;
        }
    }
    return all_epsilon;
}
void calculate_first_of_string(std::set<char> &newfirst, std::string &str,
std::map<char, std::set<char> > &first) {
    int i;
    for (i = 0; i < str.length(); i++) {
        if (isupper(str[i])) {
            // token is a nonterminal
            newfirst.insert(first[str[i]].begin(), first[str[i]].end());
            if (first[str[i]].find('<') != first[str[i]].end()) {
                // nonterminal contains epsilon in first set
                continue;
            }
            else {
                // token does not contain epsilon in first set
                break;
            }
        }
        else {
            newfirst.insert(str[i]);
        }
    }
    newfirst.erase('<');
}


void calculate_follow(std::map<char, std::set<char> > &follow,
        std::map<char, std::set<char> > &first,
        std::map<char, std::vector<std::string> > &rules,
        char S) {
```

```
    int size_before_iteration, size_after_iteration;
    std::map<char, std::vector<std::string > >::iterator i;
    follow[S].insert('$');
    size_before_iteration = 0;
    while (true) {
        for (i = rules.begin(); i != rules.end(); i++) {
            int j;
            char nonterminal = i->first;
            for (j = 0; j < (i->second).size(); j++) {
                // For every rule
                std::string wAwdash = i->second[j];
                int k;
                for (k = 0; k < wAwdash.length(); k++) {
                    if (isupper(wAwdash[k])) {
                        char A = wAwdash[k];
                        //std::cout << "{" << std::endl;
                        //std::cout << "rule: " << nonterminal << " -> "
<< wAwdash << std::endl;
                        //std::cout << "A = " << A << std::endl;
                        std::string wdash = wAwdash.substr(k + 1);
                        //std::cout << "wAwdash = " << wAwdash << std::endl;
                        //std::cout << "wdash = " << wdash << std::endl;
                        if (wdash.length() == 0) {
                            // wdash is empty
                            //std::cout <<"wdash  is empty, add follow of "
<< nonterminal  << " to follow of " << A << std::endl;
                            follow[A].insert(follow[nonterminal].begin(),
follow[nonterminal].end());
                        }
                        if (!isupper(wdash[0])) {
                            // first token in wdash is a terminal
                            //std::cout << wdash << " starts with a terminal"
<< std::endl;
                            follow[A].insert(wdash[0]);
                        }
                        if (isupper(wdash[0])) {
                            std::set<char> first_of_wdash;
                            calculate_first_of_string(first_of_wdash, wdash,
first);
                            follow[A].insert(first_of_wdash.begin(),
first_of_wdash.end());
                            //std::cout << "Adding first(" << wdash << ") = ";
                            std::set<char>::iterator l;
                            for (l = first_of_wdash.begin(); l !=
first_of_wdash.end(); l++) {
                                //std::cout << *l << ", ";
                            }
                            //std::cout << " to follow of " << A << std::endl;
                        }
                        if (is_reductible_to_empty(wdash, first)) {
                            // wdash is reductible to <
                            //std::cout << wdash << " is reductible to 0, add
follow of " << nonterminal << " to follow of " << A << std::endl;
                            follow[A].insert(follow[nonterminal].begin(),
follow[nonterminal].end());
                        }
                        //std::cout << "}" << std::endl;
                        //std::cout << "---" << std::endl;
                        //print_sets(follow, "follow");
                        //std::cout << "---" << std::endl;
                    }
                }
            }
        }
        size_after_iteration = calculate_total_size(follow);
        if (size_before_iteration == size_after_iteration) {
            // no change
            break;
        }
        else {
            // new "size before iteration"
            size_before_iteration = size_after_iteration;
        }
        //std::cout << "---" << std::endl;
        //print_sets(follow, "follow");
        //std::cout << "---" << std::endl;
        //std::cout << size_after_iteration << std::endl;
    }
    print_sets(follow, "follow");
}
// Parser Table type definition
typedef std::map<char, std::map<char, std::string>> ParsingTable;
// Function to construct the LL(1) parsing table
// Function to construct the LL(1) parsing table
ParsingTable constructParsingTable(std::map<char, std::set<char>> &first,
                                   std::map<char, std::set<char>> &follow,
                                   std::map<char, std::vector<std::string>>
&rules) {
    ParsingTable table;
    for (auto &rule : rules) {
        char nonTerminal = rule.first;
        for (auto &production : rule.second) {
            std::string currentProduction = production;
            // For each terminal in First(production)
            for (char terminal : first[nonTerminal]) {
                if (terminal == '<') { // If epsilon in First(production)
                    // For each terminal in Follow(nonTerminal)
                    for (char followSetTerminal : follow[nonTerminal]) {
                        table[nonTerminal][followSetTerminal] = production;
                    }
                } else { // If terminal is in First(production)
                    table[nonTerminal][terminal] = production;
                }
            }
            // If epsilon in First(production) and $ in Follow(nonTerminal)
            if (first[nonTerminal].find('<') != first[nonTerminal].end() &&
follow[nonTerminal].find('$') != follow[nonTerminal].end()) {
                table[nonTerminal]['$'] = production;
            }
        }
    }
    return table;
}
// Function to perform LL(1) parsing
void parseLL1(ParsingTable &table, std::string &input) {
    std::string stack = "$"; // Stack initialized with end marker
    stack.push_back(table.begin()->first); // Push start symbol to stack
    size_t inputIndex = 0;
    char stackTop;
    while (!stack.empty() && inputIndex < input.size()) {
        stackTop = stack.back();
        if (stackTop == input[inputIndex]) { // If stack top matches input
```

```
            stack.pop_back();
            inputIndex++;
        } else if (!isupper(stackTop)) { // If stack top is a terminal
            std::cerr << "Error: Mismatch between input '" << input[inputIndex]
<< "' and stack top '" << stackTop << "'\n";
            return;
        } else if (table[stackTop].find(input[inputIndex]) ==
table[stackTop].end()) { // If no production rule found
            std::cerr << "Error: No production rule found for input '"
<< input[inputIndex] << "' and stack top '" << stackTop << "'\n";
            return;
        } else { // If production rule found
            std::string production = table[stackTop][input[inputIndex]];
            stack.pop_back(); // Pop stack top
            if (production != "<") { // If production is not epsilon
                for (int i = production.size() - 1; i >= 0; --i) {
                    stack.push_back(production[i]); // Push production onto stack
in reverse order
                }
            }
        }
    }
    if (stack.empty() && inputIndex == input.size()) { // If both stack and input
are empty
        std::cout << "Input string '" << input << "' parsed successfully!\n";
    } else {
        std::cerr << "Error: Parsing failed for input '" << input << "'\n";
    }
}
int main() {
    std::map<char, std::vector<std::string> > rules;
    std::map<char, std::set<char> > terminals, first, follow;
    int err = 0;
    char S;
    std::cout << "Enter start symbol: " << std::endl;
    std::cin >> S;
    std::cout << "Enter grammar rules" << std::endl << "< represents the empty
string" << std::endl;
    while (!(err = read_rule(rules))) {
        std::cin >> std::ws;
        if (std::cin.eof()) {
            break;
        }
    }

    // check whether rules were parsed correctly
    switch (err) {
        case 1:
            std::cerr << "Terminal should be uppercase" << std::endl;
            return 1;
            break;
        case 2:
            std::cerr << "Syntax error" << std::endl;
            return 1;
            break;
        default:
            break;
    }

    int token;
    // Check whether grammar is valid
    if ((token = is_invalid(rules))) {
        if (token == -1) {
            std::cerr << "Invalid rule A -> <XXX" << std::endl;
        }
        else {
            std::cerr << "Undefined nonterminal: " << token << std::endl;
        }
        return 1;
    }

    if (rules.find(S) == rules.end()) {
        // S not a nonterminal
        std::cerr << "Invalid start symbol " << S << std::endl;
        return 1;
    }
    calculate_first(first, rules);
    std::cout << std::endl << "---" << std::endl;
    calculate_follow(follow, first, rules, S);

    ParsingTable table = constructParsingTable(first, follow, rules);
    std::string input = "abc";
    parseLL1(table, input);
    return 0;
}
```