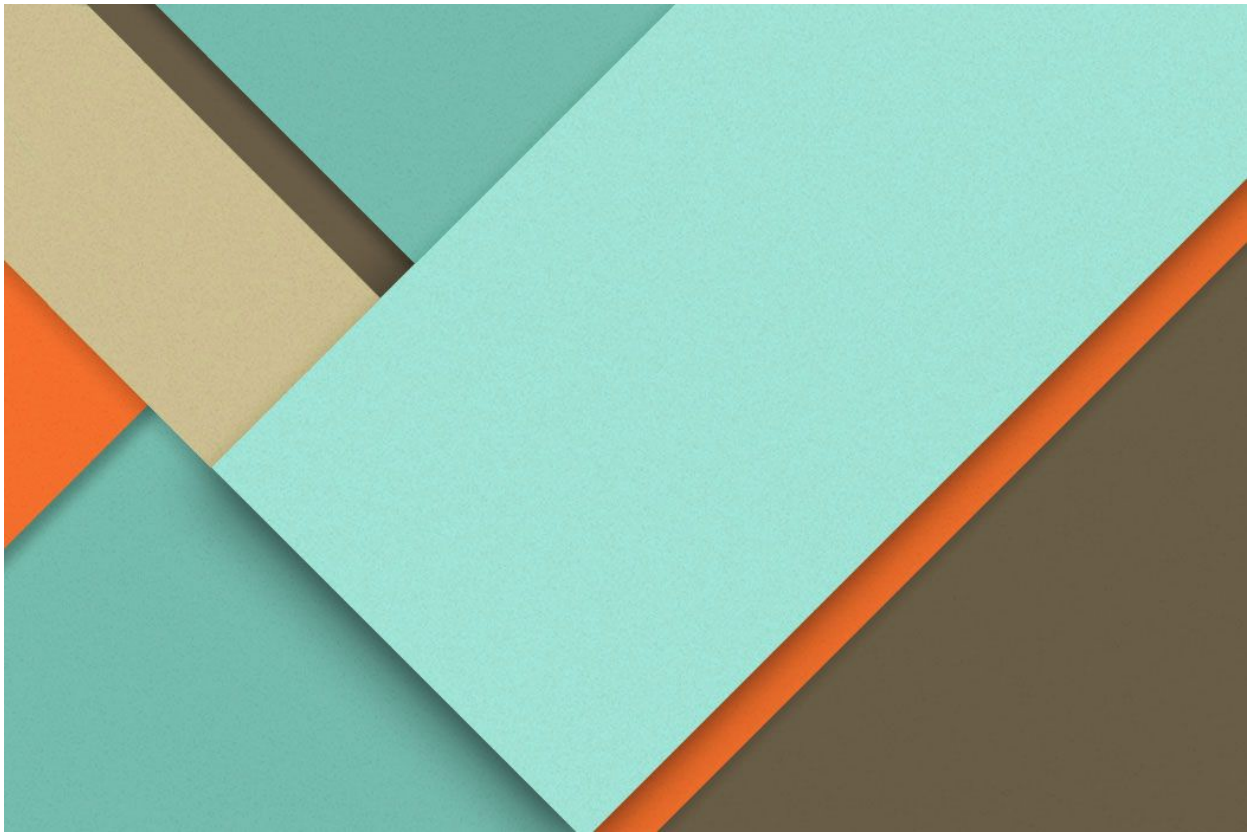

Java Compiler Phase 2

CS - 321

Ahmed Mohamed Ahmed Abdela (13)

Khaled Ali Mahmoud (25)

Mohamed Ayman Farrag (59)

Moustafa Mahmmoud (72)

Used Data Structure:

1. Rule Token:

RuleToken is a single token in a production rule which can be of different types including terminal, non terminal, Lambda, or end of input.

Such as ‘int’, METHOD_BODY, \L.

It’s used as a building unit for Production rules and contains the main logic for validating a token and categorizing it based on its format.

2. Production Rule:

ProductionRule can be abstracted to a std::vector of Rule tokens such as ‘a’ METHOD_BODY ‘c’ can be viewed as three RuleTokens.

1- Terminal RuleToken (‘a’)

2- Non terminal RuleToken(METHOD_BODY)

3- Terminal RuleToken(‘c’)

ProductionRule also contains the logic used for parsing the rules.

A Single Non terminal can have multiple production rules.

3. Predictive Table:

PredictiveTable is a data structure that provides the appropriate transition based on the current state and input.

It takes the ProductionRules and startState in the constructor to provide the required output.

4. Parse Tree Creator:

A stack which holds the current non terminal to substitute and match it using error recovery mode (Panic Mode).

Algorithms and Techniques used:

- **Left Recursion**
  - **Immediate Left Recursion**

    If(A → A α | β) Then

    {

    A → β A'

    A' → α A' | ε

    }
  - **Left Recursion**

    Nonterminal: A1….An

    for(i = 1 → n)

    {

    for(j = 1 → i-1)

    {

    Replace each production

    Ai → Aj γ

    By

    Ai → α1 γ | …. | αk γ

    Where Aj → α1 | …. | αk

    }

    }

- **First**

```
FIRST(Set set, Rule r):
    if( the first of r wasn't calculated yet):
        For each (Production p in r):
            Symbol s = first symbol in p.
            if ( s is terminal):
                set.insert(s);
            else:
                Set new = empty set;
                FIRST( new, get Rule that s names (r'));
                set = Union(set, new);
                Save new corresponding to r';
        Save set corresponding to r;
    Return first set corresponding to r;
```

- **Follow**

```
FOLLOW(Set set, Rule r):
    If( follow of r wasn't calculated yet):
        if (r.LHS = start symbol)
            set.insert('$');
        For each (Rule r' in grammar):
            For each (Production p in r'):
                For each (Symbol s in p):
                    if (s is non-terminal && s.name = r.LHS):
                        if(s is not the last symbol in p):
                            Symbol s' = Symbol following s;
                            If (s' is terminal):

                                set.insert(s');
                            else:
                                Set first = new empty set;
                                FIRST(first, get Rule the s' names (r''));
                            set = Union(set, first) // don't add lambda
                            if( lambda symbol exists in first):
                                set new = new empty set;
                                FOLLOW(new, r'');
                                set = union(set, new);
                                Save new corresponding to r'';
                        else:
                            Set new = new empty set;
                            FOLLOW(new, get non-terminal symbol naming r');
                            set = Union(set,new);
        Save set corresponding to r;
    Return follow set corresponding to r;
```

- **Parsing Table Construction**

For each production rule A → α of a grammar G

For each terminal a in First(α)

➔ Add A→ α to M[A,a]

If(ε is in FIRST(α))

➔ For each terminal a in Follow(A) add A→ α to M[A,a]

If(ε is in FIRST(α) and $ in Follow(A))

➔ add A→ α to M[A,$]

All other undefined entries of the parsing table are error entries.

Parsing Table:

Table is provided in the project as it's dynamically generated.

Comments about used tools:

https://cyberzhg.github.io/toolbox/left_fact

https://cyberzhg.github.io/toolbox/left_rec

It was used to test the that the process of left factoring and elimination of left recursion was successful against multiple test cases.

Explanation of Functions:

```
LL1Grammar ParserRulesReader::getLL1Grammar(std::ifstream* input_file, std::ofstream* modified_rules) {
    if (!input_file->is_open()) {
        std::cerr << INVALID_INPUT_FILE_MESSAGE << std::endl;
        exit(0);
    }
    auto rules = readRules(input_file);
    eliminateLeftRecursion(rules.rule_table);
    leftFactorGrammar(rules.rule_table);
    printRulesTable(rules.rule_table, *modified_rules);
    return rules;
}

LL1Grammar ParserRulesReader::readRules(std::ifstream* input) {
    std::vector<std::string> file_lines = readFile(input);
    std::set<std::string> rules_identifier = extractRuleIdentifiers(file_lines);
    return parseRules(rules_identifier, file_lines);
}
```

1. First the Grammar is read line by line, validated for the assumptions stated at the end of the report and converted into ProductionRule objects which are then used throughout the project.
2. ProductionRules are eliminated from left recursion.
3. ProductionRules are left factored.
4. New grammar that had been left factored and eliminated from left recursion is printed to ‘ll1_grammar.txt’

```
void ParserRulesReader::eliminateLeftRecursion(std::map<std::string, std::vector<ProductionRule> >& rule_table) {
    std::vector< std::vector<ProductionRule> > newly_added_states;
    int new_state_counter = 0;
    for(auto it_i = rule_table.begin(); it_i != rule_table.end(); it_i++) {
        for(auto it_j = rule_table.begin(); it_j != it_i; it_j++) {
            std::vector<ProductionRule> current_productions = it_i->second;
            std::vector<ProductionRule> new_rules;
            for(ProductionRule prod_rule : current_productions) {
                if(prod_rule.canLeftImmedSubstitute(it_j->first)) {
                    std::vector<ProductionRule> substituted_rules = prod_rule.leftImmedSubstitute(it_j->second);
                    new_rules.insert(std::end(new_rules), std::begin(substituted_rules), std::end(substituted_rules));
                } else {
                    new_rules.push_back(prod_rule);
                }
            }
            it_i->second = new_rules;
        }
        std::vector<ProductionRule> new_state_rules = leftImmedEliminate(it_i->second, it_i->first, new_state_counter);
        if(new_state_rules.size() != 0) {
            new_state_counter++;
            newly_added_states.push_back(new_state_rules);
        }
    }
    for(size_t i = 0; i < newly_added_states.size(); i++) {
        rule_table[ParserRulesReader::createStateName(i)] = newly_added_states[i];
    }
}
```

The Algorithm used in the lecture was utilized to eliminate left recursion.

It sorts the States on a lexicographical ordering then substitute all the immediate left symbols with the the states that have lower lexicographical order and then eliminates Immediate left recursion if it exists.

In case the immediate left recursion can't be eliminated because all the production rule will have the same state as the first symbol, It prints an error and exits.

```
void ParserRulesReader::leftFactorGrammar(std::map<std::string, std::vector<ProductionRule> >& rule_table) {
    std::vector<std::string> to_be_factored_rules;
    for(auto rule_it : rule_table) {
        to_be_factored_rules.push_back(rule_it.first);
    }
    for(std::string rule : to_be_factored_rules) {
        leftFactorProduction(rule, rule_table);
    }
}

void ParserRulesReader::leftFactorProduction(std::string rule_name, std::map<std::string, std::vector<ProductionRule> >& rule_table) {
    std::vector<ProductionRule> to_be_factored_rules = rule_table[rule_name];
    std::map<RuleToken, std::vector<ProductionRule> > production_rule_prefix_map;
    std::vector<ProductionRule> new_rules;
    for(ProductionRule prod_rule : to_be_factored_rules) {
        if(prod_rule.getTokenCount() != 0) {
            production_rule_prefix_map[prod_rule.getToken(0)].push_back(prod_rule);
        } else {
            new_rules.push_back(ProductionRule({RuleToken(Constants::LAMBDA)}));
        }
    }
    for(auto production_rule_prefix_it : production_rule_prefix_map) {
        if(production_rule_prefix_it.second.size() == 1) {
            new_rules.push_back(production_rule_prefix_it.second[0]);
            continue;
        }
        size_t prefix_length = ProductionRule::getCommonPrefixTokenCount(production_rule_prefix_it.second);
        std::vector<RuleToken> prefix_tokens = production_rule_prefix_it.second[0].getPrefixTokens(prefix_length);
        std::string new_state_name = ParserRulesReader::generateState();
        prefix_tokens.push_back(RuleToken(new_state_name));
        new_rules.push_back(ProductionRule(prefix_tokens));
        for(ProductionRule& prod_rule : production_rule_prefix_it.second) {
            prod_rule.popTokens(prefix_length);
        }
        rule_table[new_state_name] = production_rule_prefix_it.second;
        leftFactorProduction(new_state_name, rule_table);
    }
    rule_table[rule_name] = new_rules;
}
```

To left factoring, Production rules are grouped into buckets based on the first Token in each production rule. In a DFS like technique, The longest common prefix is computed in each bucket and the remaining is substituted and the same function is called with the substituted partition of ProductionRule.

```
std::unordered_set<std::string> PredictiveTable:: getFirst(std::string state) {
    if(!ll1_grammar.count(state))
        std::cerr<< "Undefined token\n", exit(0);

    if(first.count(state))
        return first[state];

    std::unordered_set<std::string> &cur_first = first[state];
    for(ProductionRule &pr: ll1_grammar[state]) {
        std::vector<RuleToken> tokens = pr.getTokens();
        bool has_lambda = true;
        for(RuleToken &r : tokens) {
            if(r.getType() == RuleTokenType::NON_TERMINAL) {
                std:: unordered_set<std::string> new_first = getFirst(r.getValue());

                for(std::string s : new_first) {
                    if(s == Constants::LAMBDA)
                        continue;
                    if(cur_first.count(s))
                        std::cerr << "Multiple rules on the same input\n", exit(0);

                    cur_first.insert(s);
                    table[state][s] = pr;
                }

                has_lambda &= new_first.count(Constants::LAMBDA);
                if(!new_first.count(Constants::LAMBDA)) break;
            }
            else {
                if(!checkTerminals(r, cur_first, state, pr, TYPE::FIRST)) {
                    has_lambda = false;
                    break;
                }
            }
        }
        if(has_lambda){
            cur_first.insert(Constants::LAMBDA);
            std::unordered_set<std::string> cur_follow = getFollow(state);
            for(std::string s: cur_follow) {
                if(table[state].count(s) && table[state][s].getTokens() != pr.getTokens())
                    std::cerr << "Multiple rules on the same input\n", exit(0);
                table[state][s] = pr;
            }
        }
    }
    return cur_first;
}
```

Function to compute the first set of given non terminal
Using rules in the lecture by iterating over production rules the state is the LHS and iterating over all tokens from left to right and will break if any token in them doesn't has LAMBDA else union first of this token with cur_first (first of state).

```
std::unordered_set<std::string> PredictiveTable:: getFollow(std:: string state) {
    if(!ll1_grammar.count(state))
        std::cerr << "Undefined token\n", exit(0);

    if(follow.count(state)) return follow[state];

    std::unordered_set<std::string> &cur_follow = follow[state];
    calcRHSFollow(cur_follow, state);
    if(state == start_state)
        cur_follow.insert(Constants::END_OF_INPUT);

    for(ProductionRule &pr: ll1_grammar[state]) {
        std::vector<RuleToken> tokens = pr.getTokens();
        std::reverse(tokens.begin(), tokens.end());
        for(RuleToken &r : tokens) {
            if(r.getType() == RuleTokenType::NON_TERMINAL) {
                std::unordered_set<std::string> new_follow = getFollow(r.getValue());
                cur_follow.insert(new_follow.begin(), new_follow.end());
                if(!getFirst(r.getValue()).count(Constants::LAMBDA)) break;
            }
            else
                checkTerminals(r, cur_follow, state, pr, TYPE::FOLLOW);
        }
        reverse(tokens.begin(), tokens.end());

    }

    return cur_follow;
}
```

--Functions used from outer classes to deal with the predictive table
Get transition type retrieves the type of the transition of given non terminal at given input, Types are sync, error or legal (not sync and not error)
Get transition retrieves the transition of the non terminal if and only if its type is legal

```
void ParseTreeCreator::createTable(std::ofstream* output_file) {
    RuleToken non_terminal(start_state), dollar(Constants::END_OF_INPUT);
    parse_tree_stack.push(dollar), parse_tree_stack.push(non_terminal);
    print_non_terminal.push_front(non_terminal);
    while (!parse_tree_stack.empty()) {
        RuleToken top_of_stack = parse_tree_stack.top();
        parse_tree_stack.pop();
        if (top_of_stack.getType() == RuleTokenType::END_OF_INPUT) {
            if (tokens.nextToken() != Constants::END_OF_INPUT)
                error = true;
            parse_tree_stack.push(top_of_stack);
            tokens.getNextToken();
            continue;
        } else if (top_of_stack.getType() == RuleTokenType::LAMBDA_TERMINAL) {
            continue;
        } else if (top_of_stack.getType() == RuleTokenType::NON_TERMINAL) {
            handleNonTerminal(top_of_stack, output_file);
        }
        else if (top_of_stack.getType() == RuleTokenType::TERMINAL) {
            if (top_of_stack.getValue() != tokens.nextToken())
                error = true;
            else
                tokens.getNextToken();
        }
    }
}
```

Algorithm used to simulate Panic mode recovery for LL(1):
-- First we have the start state at the top of the stack and we have tokens read from file to be matched.

-- We compare top of stack with the needed token and match them if we can.
-- If we can't match, we try to either substitute the non termal with proper tokens or we have an error.

-- Panic mode deals with error as follow:

```
void ParseTreeCreator::handleNonTerminal(RuleToken top_of_stack, std::ofstream* output_file) {
    if (predictive_table.getTransitionType(top_of_stack.getValue(),
                tokens.nextToken()) == TransitionType::LEGAL) {
        vector<RuleToken> transitions = predictive_table.getTransition(top_of_stack.getValue(),
            tokens.nextToken());
        for (int i = transitions.size() - 1; i >= 0; --i)
            parse_tree_stack.push(transitions[i]);
        substituteNonTerminal(transitions);
        print(output_file);
    } else if (predictive_table.getTransitionType(top_of_stack.getValue(),
        tokens.nextToken()) == TransitionType::SYNC) {
        error = true;
    } else {
        error = true;
        parse_tree_stack.push(top_of_stack);
        tokens.getNextToken();
    }
}
```

If ‘sync’ we discard the non terminal

If ‘error’ we discard the current token

If top of stack is a terminal node we insert it to tokens as it is a missing token from input.

-- This algorithm Prints the state every moment we substitute a non terminal node

```
void ParseTreeCreator::print(std::ofstream* output_file) {
    for (int i = 0; i < (int) print_terminal.size(); ++i)
        *output_file << print_terminal[i].getValue() << " ";
    for (int i = 0; i < (int) print_non_terminal.size(); ++i)
        *output_file << print_non_terminal[i].getValue() << " ";
    *output_file << std::endl;
}
```

Assumptions:

1. States or Production rules must follow a specific format of alphanumeric structure starting with an alphabet.
2. States or Production rules must never use two name formats ‘State_x’, ‘Generated_State_x’ where x is any positive number as these names are used to generate fake states to eliminate left recursion or left factor production rules.
3. Any reserved symbol such as ‘|’, single quotes or ‘=’ must be preceded with a backslash.

   I.e # A = ‘\=’ ‘\|’ ‘\”

4. Lambda is described using \L without using single quotes.

COPIED

There is a space delimiting # and LHS in the grammar file.

Line should start by # or | in the grammar file.

There is a space delimiting LHS and = in the grammar file.

There is a space delimiting = and the 1st production in the grammar file.

There is a space delimiting production and | in the grammar file.

There is a space delimiting | and production in the grammar file.

No two symbols have the same name where one is terminal and the other is nonterminal..

Any nonterminal symbol must have a rule that defines it.