Compilers  
Parser Generator

22/4/2020

# data structures

## Syntactic Term

Syntactic term is an inheritance class used to represent the non-terminal terms and it consists of:

* Vector of Production Rules for this term.
* Unordered set of strings for the first
* Unordered set of strings for the follow

## Production term

Production term is a class used to represent terminal and non-terminal terms and it consists of:

* String to save the name of this term.
* Production term type either terminal or non-terminal.

## Production Rule

Production rule is a class used to represent rules and it consists of:

* Vector of pointers on Production terms to save all terms in RHS of the rule.
* Pointer on Syntactic term which is in the LHS of the rule.

## Parsing Table

It is a class to calculate first, follow and parsing table:

* First: A map its key is pointer on the Syntactic term and its value is unordered set of strings which contains the terminals of the first set.
* First: A map its key is pointer on the Syntactic term and its value is unordered set of strings which contains the terminals of the follow set.
* A map its key is pointer on the Syntactic term and its value is another map, this map of key string “the terminal” and the value is a production rule.

## We defined the following helping enums:

* + - * ProductionTermType: ***Terminal***, ***NonTerminal***.

# Algorithms

## Eliminate left recursion and left factoring of CFG 🡨 need to update

|  |  |
| --- | --- |
|  | To convert regular expressions to NFA; Thompson’s construction algorithm is used.  So, there some main procedures to do this construction.   1. Read the Lexical Rules from the file and parsing them using **ReadLexicalRulesFile** class. 2. Convert the parsing Lexical Rules to Postfix expressions using **LexicalRuleBuilder** class. 3. Build the NFA for each Postfix Lexical Rule using **Builder** class.   This class applies the Thompson’s construction algorithm Rules such as |

## Compute First and Follow

|  |  |
| --- | --- |
|  | To Calculate first of a non-terminal term we do this steps:   * Loop on all productions for this term * Cheek if this production rule is epsilon or not i.e: term 🡪 EPS if yes add eps to answer and continue to the next production rule. * Get the first term in the left side there is to cases - term is terminal: then add it to the set of the result - term is non-terminal: add the first of this term to the result set, if it still didn’t computed then we call recursively the function. * If there is an epsilon in the first of the first non-terminal, then we check the term after it and so on. * Each string we get and put in the result set, it’s added instantaneously to the table with the production rule.   unordered\_set<string> ParsingTable::getFirst(SyntacticTerm\* non\_terminal) {  unordered\_set<string> res;  if (first.find(non\_terminal) == first.end()) {  vector<ProductionRule \*> x = non\_terminal->getProductions();  /// loop on each production rule and get from it first  for (auto productionRule:x) {  unordered\_set<string> onePR;  if (productionRule->isEpsilon()) {  res.insert("EPS");  continue;  }  int index = 0;  auto \*firstTerm = (SyntacticTerm \*) productionRule->getTerms().at(index);  /// if symbol is terminal then add to the list   if (firstTerm->getType() == *Terminal*) {  if (table.find(non\_terminal) == table.end()){  map<std::string, struct ProductionRule> newchar;  newchar.insert(pair<std::string, struct ProductionRule>(firstTerm->getName(), \*productionRule));  table.insert(pair<SyntacticTerm \*, map<std::string, struct ProductionRule>>(non\_terminal,newchar));  } else {  if (table.find(non\_terminal)->second.find(firstTerm->getName()) != table.find(non\_terminal)->second.end()){  table.find(non\_terminal)->second.find(firstTerm->getName())->second = \*productionRule;  } else {  table.find(non\_terminal)->second.insert(  pair<std::string, struct ProductionRule>(firstTerm->getName(), \*productionRule));  }  }  res.insert(firstTerm->getName());  ///else if symbol is non-terminal then compute its first then add it  } else {  unordered\_set<string> temp;  /// if the first is already computed add it  if (first.find(firstTerm) != first.end()) {  temp = first.at(firstTerm);  ///else if it's not computed before then recursive call the function  } else {  temp = getFirst(firstTerm);  }  /// handling special case for having epsilon at first of the first non-terminal terms.  while (temp.find("EPS") != temp.end()) {  temp.erase("EPS");  res.insert(temp.begin(), temp.end());  onePR.insert(temp.begin(), temp.end());  index++;  if (index < productionRule->getTerms().size()) {  auto \*nextTerm = (SyntacticTerm \*) productionRule->getTerms().at(index);  temp = getFirst(nextTerm);  } else {  temp.insert("EPS");  break;  }  }  res.insert(temp.begin(), temp.end());  onePR.insert(temp.begin(), temp.end());  }  for (const auto& c:onePR){  if (table.find(non\_terminal) != table.end()){  if (table.find(non\_terminal)->second.find(c) != table.find(non\_terminal)->second.end()){  table.find(non\_terminal)->second.find(c)->second = \*productionRule;  } else {  table.find(non\_terminal)->second.insert(pair <std::string, struct ProductionRule> (c, \*productionRule));  }  } else {  map<std::string, struct ProductionRule> newchar;  newchar.insert(pair<std::string, struct ProductionRule>(c, \*productionRule));  table.insert(pair<SyntacticTerm \*, map<std::string, struct ProductionRule>>(non\_terminal,newchar));  }  }  }  first.insert(pair<SyntacticTerm \*, unordered\_set<string>>(non\_terminal, res));  non\_terminal->setFirst(res);  return res;  } else {  non\_terminal->setFirst(first.at(non\_terminal));  return first.at(non\_terminal);  } }  To Calculate follow of all non-terminal term we do this steps:   * Loop on all non-terminal term we have. * Loop on all production rules for each term. * Loop on all terms in each production rule twice -forward: to know each term id followed by what, saving terminals in a set and the non-terminals in another set till the time it will be calculated - backward: to get cases for the epsilon in the last element. * At the end all non-terminal sets are cleared and but instead it the follow of this terms.   void ParsingTable::setFollowTable(vector<SyntacticTerm\*> non\_terminal) {  map<SyntacticTerm\*, unordered\_set<SyntacticTerm\*>> nonterm\_follow;  /// setting the sets foe follow results  for (auto item:non\_terminal) {  unordered\_set<string> newSet;  unordered\_set<SyntacticTerm\*> newSetT;  if (non\_terminal.at(0) == item){ /// If S is the start symbol ⎝ $ is in FOLLOW(S)  newSet.insert("$");  }  follow.insert(pair<SyntacticTerm\*, unordered\_set<string>>(item,newSet));  nonterm\_follow.insert(pair<SyntacticTerm\*, unordered\_set<SyntacticTerm\*>>(item,newSetT));  }  /// loop on all non terminals  for (auto item:non\_terminal){  /// loop on all productions of each non-terminal item  for (auto p:item->getProductions()) {  vector<ProductionTerm \*> terms =p->getTerms();  /// loop on terms of each production rule twice: forward and backward  for (int i = 1;i <terms.size();i++){   if (terms.at(i-1)->getType() == *NonTerminal*) {  if (terms.at(i)->getType() == *Terminal*) {  if (follow.find((SyntacticTerm \*) terms.at(i - 1)) != follow.end()) {  follow.find((SyntacticTerm \*) terms.at(i - 1))->second.insert(terms.at(i)->getName());  } else {  unordered\_set<string> newSet;  newSet.insert(terms.at(i)->getName());  follow.insert(pair<SyntacticTerm\*, unordered\_set<string>>((SyntacticTerm \*) terms.at(i - 1),newSet));  }  } else { /// if A-> aBb is a production rule ⎝ everything in FIRST(b) is FOLLOW(B) except EPS  unordered\_set<string> temp ;  if (first.find((SyntacticTerm \*)terms.at(i)) != first.end()) {  temp = first.find((SyntacticTerm \*)terms.at(i))->second;  } else {  temp = getFirst((SyntacticTerm \*)terms.at(i));  }  if (temp.find("EPS") != temp.end()) {  temp.erase(temp.find("EPS"));  }  if (follow.find((SyntacticTerm \*) terms.at(i - 1)) != follow.end()) {  follow.find((SyntacticTerm \*) terms.at(i - 1))->second.insert(temp.begin(), temp.end());  } else {  unordered\_set<string> newSet;  newSet.insert(temp.begin(), temp.end());  follow.insert(pair<SyntacticTerm\*, unordered\_set<string>>((SyntacticTerm \*) terms.at(i - 1),newSet));  }  }  }  }  /// loop backward to get cases for the epsilon in last elements  if (terms.at(terms.size()-1)->getType() == *NonTerminal* && terms.at(terms.size()-1) != item){  nonterm\_follow.find((SyntacticTerm \*) terms.at(terms.size()-1))->second.insert((SyntacticTerm \*) item);  }  int n = terms.size();  n--;  for (int i = n;i > 0;i--){  if (terms.at(i)->getType() == *NonTerminal* && terms.at(i-1)->getType() == *NonTerminal*){  if (((SyntacticTerm \*) terms.at(i))->isDerivingToEpsilon() && ( terms.at(i - 1) != item)){  nonterm\_follow.find((SyntacticTerm \*) terms.at(i - 1))->second.insert(item);  } else {  break;  }  } else {  break;  }  }  }  }  /// to finalize the follow results and remove non-terminal from it  finalizingfollow(nonterm\_follow); }  void ParsingTable::finalizingfollow(map<SyntacticTerm \*, unordered\_set<SyntacticTerm \*>> nonterm\_follow) {  int i = 0;  int times = 0;  do {  times ++;  i = 0;  for (auto item:nonterm\_follow) {  if (!item.second.empty()) {  i++;  /// eliminate non-terminal from other elements  for (auto item2:item.second) {  if (nonterm\_follow.find(item2)->second.empty() && item.second.find(item2) != item.second.end()){  follow.find(item.first)->second.insert(  follow.find(item2)->second.begin(), follow.find(item2)->second.end());  item.second.erase(item.second.find(item2));  nonterm\_follow.find(item.first)->second = item.second;  }  }  }  }  } while (i != 0 && times <= nonterm\_follow.size()); }  void ParsingTable::finalizingfollow(map<SyntacticTerm \*, unordered\_set<SyntacticTerm \*>> nonterm\_follow) {  int i = 0;  int times = 0;  do {  times ++;  i = 0;  for (auto item:nonterm\_follow) {  if (!item.second.empty()) {  i++;  /// eliminate non-terminal from other elements  for (auto item2:item.second) {  if (nonterm\_follow.find(item2)->second.empty() && item.second.find(item2) != item.second.end()){  follow.find(item.first)->second.insert(  follow.find(item2)->second.begin(), follow.find(item2)->second.end());  item.second.erase(item.second.find(item2));  nonterm\_follow.find(item.first)->second = item.second;  }  }  }  }  } while (i != 0 && times <= nonterm\_follow.size()); } |

## Constructing Parsing Table

|  |  |
| --- | --- |
|  | Steps:   * Loop on all non-terminals to get calculate all first values and add it to the parsing table. * Calculate follow for all non-terminals. * Loop on follows we have -If the term is driving to Epsilon then add the production term 🡪 EPS to cells from the set of follow in the table. --- if this cell is already have a production rule set ambiguity to true. -If it’s not driving to Epsilon then add synch to the cells. * Then return table.   void ParsingTable::settingFirstANDFollow(const vector<SyntacticTerm \*>& non\_terminal) {  auto\* synch = new ProductionRule();  if (!cons) {  for (auto i:non\_terminal) {  getFirst(i);  }  setFollowTable(non\_terminal);  for (auto i:non\_terminal) {  i->setFollow(follow.find(i)->second);  /// add sync to table  if (i->isDerivingToEpsilon()){  for (auto \*pr:i->getProductions()){  if (pr->isEpsilon()){  for (const auto& c:i->getFollow()){  if (table.find(i)->second.find(c) != table.find(i)->second.end()){ // table.find(i)->second.find(c)->second = \*pr;  amb = true;  } else {  table.find(i)->second.insert(pair <std::string, struct ProductionRule>(c,\*pr));  }  }  break;  }  }  } else {  for (const auto& c:i->getFollow()) {  if (!(table.find(i)->second.find(c) != table.find(i)->second.end())){  table.find(i)->second.insert(pair <std::string, struct ProductionRule>(c,\*synch));  }  }  }  }  cons = true;  } }  map<SyntacticTerm \*, map<std::string, struct ProductionRule>> ParsingTable::getTable(const vector <SyntacticTerm\*>& non\_terminal) {  if(non\_terminal.empty()){  return table;  }  settingFirstANDFollow(non\_terminal);   return table; } |

## Left Most Derivation 🡨 need to update

|  |  |
| --- | --- |
|  | We used Backtracking technique in Scanner that is the Scanner starts moving with the  input in the graph until reaching an invalid state or the input ends then it backtracks until  reaching the last acceptance state according to Maximal Munch.  Constructor of parser:   * Read the lexical file. * Get the productions and remove left recursion and left factoring. * Printing table if needed.      * There is vector of strings called to have errors. * There is vector of vector of production term called derivations to have derivations.   Loop on all the left most derivations to save it in output which is vector of strings.    Bool function that return false if the grammar is ambiguous or if there is an error in the file or the productions, return true otherwise.   * Push in stack the dollar sign then the start production. * Token t has the input tokens.      * Check if the stack is empty then loop and do the algorithm or steps of left most derivation. * Check if the stack top is terminal: * If so, check if the input token equals the stack top, pop the stack then get the next token. * If the token is not equal the stack, check if it is epsilon, if so, then pop. * Else, save this error as missing error. * If the stack top is not terminal. * If the stack top under the token goes to nothing, then pop the stack and save the error discard and get the next token. * If the stack top under the token goes to production, check if it goes to sync, calls handle derivation then pop the stack. * Else, pop the stack then calls handle derivation     Check if the stack and errors are empty, then accept the grammar, else not accept the grammar.    Handle derivation that replace one non terminal in the stack with its production. |

**TRANSITION Diagrams and Parsing Tables** 🡨 need to update

# Comments about used tools 🡨 need to update

# Explanation of functions of all phases 🡨 need to update

## Priorities of Accepted States:

We assumed that accepted states have the following priorities from high to low (punctuation >> keywords >> regular expression) which is followed by most of the popular programming languages.

## All Reserved character must be escaped:

Any reserved character should always be escaped before usage, otherwise it would be considering it as an operation if suitable.

Reserved characters are {‘+’, ‘-’, ‘\*’, ‘|’, ‘(‘, ‘)’}

# assumptions 🡨 need to update

## Priorities of Accepted States:

We assumed that accepted states have the following priorities from high to low (punctuation >> keywords >> regular expression) which is followed by most of the popular programming languages.

## All Reserved character must be escaped:

Any reserved character should always be escaped before usage, otherwise it would be considering it as an operation if suitable.

Reserved characters are {‘+’, ‘-’, ‘\*’, ‘|’, ‘(‘, ‘)’}

## Grouping Characters:

Characters are grouped by their numerical values, so saying ‘a-z’, would lead to considering any character between the numeric values of a to z to be included. That would lead to us rejecting any character group where the ending character has a numeric value less than the starting character.

## Regular Expressions Definition:

Regular definitions can be used in multiple regular expressions or definitions, however

regular expressions can’t be reused to define another regular expression or definition. Also, a regular definition will only be replaced if it was already defined before being referenced, or it will consider it a normal word.

# Team members

|  |  |
| --- | --- |
| Name | ID |
| **Arsany Atef Abdo** | **10** |
| **Kirellos Malak Habib** | **35** |
| **Michael Said Beshara** | **38** |
| **Yomna Gamal El-Din Mahmoud** | **63** |