**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение высшего образования  
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ имени Н.Э.БАУМАНА  
(национальный исследовательский университет)»**

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

**Лабораторная работа № 8**

“Множество FIRST для РБНФ”

по дисциплине «Конструирование компиляторов»

Вариант 1 (JAVA)

Работу выполнил

студент группы ИУ9-62Б

Сербин Денис

Москва, 2022

# 

# **1. Цель работы**

Целью данной работы является изучение алгоритма построения множеств FIRST для расширенной формы Бэкуса-Наура.

**2. Исходные данные**

В данной лабораторной работе требуется разработать программу, которая по описанию грамматики, записанному на входном языке в РБНФ, строит множества FIRST для всех нетерминалов грамматики.

# В качестве входного языка должен выступать язык представления правил грамматики, варианты лексики и синтаксиса которого можно восстановить по примерам.

# **3. Задание**

Выполнение данной лабораторной работы состоит из следующих этапов:

1. Составление описаний лексической структуры и грамматики входного языка на основе примера.

2. Разработка лексического анализатора для входного языка.

3. Разработка синтаксического анализатора для входного языка методом рекурсивного спуска.

4. Реализация алгоритма вычисления множества FIRST для всех нетерминальных символов грамматики.

Отметим, что парсер входного языка должен выдавать сообщения об обнаруженных ошибках, включающие координаты ошибки. Восстановление при ошибках реализовывать ненужно.

В качестве языков реализации разрешается использовать любой язык с поддержкой рекурсии.

**4. Индивидуальный вариант**

# объявления тоже надо проверять

non-terminal E, T, F;

terminal '+', '-', '\*', '/', (',')', n;

E ::= T (('+' | '-') T )\*;

T ::= F((\* |/') F )\*;

F ::= n '-' F | '( Е ')';

**5. Реализация**

Грамматика в РБНФ:

S ::= DN DT RLST?

DN ::= "non-terminal" net NLST? ';'

NLST ::= ',' net NLST?

DT ::= "terminal" tern TLST? ';'

TLST ::= ',' term TLST?

RLST ::= net "::=" R\* ';' RLST?

R ::= RE ('|' R)\*

RE ::= (net|term|P) RE1

RE1 ::= ('+'|'?'|'\*')? RE

P ::= '(' R ')'

// S ::= DN DT RLST?  
 **private void** parseS(ParserTags.PARSE\_MODE mode) **throws** Exception{  
 **int** index = **this**.parseDN(mode, 0);  
 index = **this**.parseDT(mode, index);  
 **this**.parseRLST(mode, index);  
 }  
  
 // DN ::= "non-terminal" net NLST? ';'  
 **private int** parseDN(ParserTags.PARSE\_MODE mode, **int** index) **throws** Exception {  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*NONTERMINALSIGN*) {  
 **throw new** Exception(String.*format*("expected '::=' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 Rule rule = **null**;  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*NONTERMINAL*) {  
 **throw new** Exception(String.*format*("expected nonterminal token at position:%s", **this**.chain.get(index).toString()));  
 }  
 **if** (mode == ParserTags.PARSE\_MODE.*INITIAL*) {  
 Nonterminal newN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 **this**.addNonterminal(newN);  
 }  
 **if** (mode == ParserTags.PARSE\_MODE.*INITIAL*) {  
 Nonterminal newRuleN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 **this**.addRule(newRuleN);  
 }  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 Nonterminal newRuleN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 rule = rules.get(rulesMap.get(newRuleN));  
 }  
 index++;  
  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
  
 index = **this**.parseNLST(mode, index, rule);  
 }  
  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*DOT*) {  
 **throw new** Exception(String.*format*("expected ';' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **return** index;  
 }  
  
 // NLST ::= ',' net NLST?  
 **private int** parseNLST(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception{  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
 **throw new** Exception(String.*format*("expected = token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*NONTERMINAL*) {  
 **throw new** Exception(String.*format*("expected nonterminal token at position:%s", **this**.chain.get(index).toString()));  
 }  
 **if** (mode == ParserTags.PARSE\_MODE.*INITIAL*) {  
 Nonterminal newRuleN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 **this**.addRule(newRuleN);  
 }  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 Nonterminal newRuleN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 rule = rules.get(rulesMap.get(newRuleN));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
 **return this**.parseNLST(mode, index, rule);  
 }  
 **return** index;  
 }  
  
 // DT ::= "terminal" tern TLST? ';'  
 **private int** parseDT(ParserTags.PARSE\_MODE mode, **int** index) **throws** Exception {  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*TERMINALSIGN*) {  
 **throw new** Exception(String.*format*("expected '::=' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 Rule rule = **null**;  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*TERMINAL*) {  
 **throw new** Exception(String.*format*("expected nonterminal token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
 index = **this**.parseTLST(mode, index, rule);  
 }  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*DOT*) {  
 **throw new** Exception(String.*format*("expected ';' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **return** index;  
 }  
  
 // TLST ::= ',' term TLST?  
 **private int** parseTLST(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception{  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
 **throw new** Exception(String.*format*("expected '::=' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*TERMINAL*) {  
 **throw new** Exception(String.*format*("expected nonterminal token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*COMMA*) {  
 **return this**.parseTLST(mode, index, rule);  
 }  
 **return** index;  
 }  
  
 // RLST ::= net "::=" R\* ';' RLST?  
 **private int** parseRLST(ParserTags.PARSE\_MODE mode, **int** index) **throws** Exception {  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*NONTERMINAL*) {  
 **throw new** Exception(String.*format*("expected nonterminal token at position:%s", **this**.chain.get(index).toString()));  
 }  
 Rule rule = **null**;  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 Nonterminal newRuleN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 rule = rules.get(rulesMap.get(newRuleN));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*EQ*) {  
 **throw new** Exception(String.*format*("expected '::=' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 index = **this**.parseR(mode, index, rule);  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*DOT*) {  
 **throw new** Exception(String.*format*("expected ';' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*END*) {  
 **return** index;  
 }  
 index = parseRLST(mode, index);  
 **return** index;  
 }  
  
 // R ::= RE ('|' R)\*  
 **private int** parseR(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception{  
 index = **this**.parseRE(mode, index, rule);  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*DELIMITER*) {  
 index++;  
 **return this**.parseR(mode, index, rule);  
 }  
 **return** index;  
 }  
  
 // RE ::= (net|term|P) RE1  
 **private int** parseRE(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception {  
 ParserTags.PARSE\_MODE modeTimes = mode;  
 ParserTags.PARSE\_MODE modeSelf = mode;  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*NONTERMINAL*) {  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 Nonterminal newN = **new** Nonterminal(**this**.chain.get(index).getAttribute());  
 Rule add = **this**.rules.get(**this**.rulesMap.get(newN));  
 **if** (add.hasEps()) {  
 rule.addToFirst(add.getFIRSTwoEps());  
 modeSelf = ParserTags.PARSE\_MODE.*CALCULATE*;  
 } **else** {  
 rule.addToFirst(add.getFIRST());  
 modeSelf = ParserTags.PARSE\_MODE.*SKIP*;  
 }  
 }  
 index++;  
 index = **this**.parseRE1(modeTimes, index, rule);  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if** (rule.hasEps()) {  
 rule.removeEps();  
 modeSelf = ParserTags.PARSE\_MODE.*CALCULATE*;  
 }  
 }  
 index = **this**.parseRE(modeSelf, index, rule);  
 **return** index;  
 } **else if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*TERMINAL*) {  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if** (!rule.getFIRST().contains(**new** Terminal(**this**.chain.get(index).getAttribute()))) {  
 rule.addToFirst(**new** Terminal(**this**.chain.get(index).getAttribute()));  
 rule.setChangedFirst(**true**);  
 } **else**{  
 rule.setChangedFirst(**false**);  
 }  
 modeTimes = ParserTags.PARSE\_MODE.*CALCULATE*;  
  
 }  
 index++;  
 index = **this**.parseRE1(modeTimes, index, rule);  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if** (rule.hasEps()) {  
 **if**(rule.isChangedFirst()) {  
 rule.removeEps();  
 }  
 modeSelf = ParserTags.PARSE\_MODE.*CALCULATE*;  
  
 } **else** {  
 modeSelf = ParserTags.PARSE\_MODE.*SKIP*;  
 }  
 }  
 index = **this**.parseRE(modeSelf, index, rule);  
 **return** index;  
 } **else if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*BR\_OPEN*) {  
 index++;  
 index = **this**.parseP(mode, index, rule);  
 index = **this**.parseRE1(mode, index, rule);  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if** (rule.hasEps()) {  
 rule.removeEps();  
 modeSelf = ParserTags.PARSE\_MODE.*CALCULATE*;  
  
 } **else** {  
 modeSelf = ParserTags.PARSE\_MODE.*SKIP*;  
 }  
 }  
 index = **this**.parseRE(modeSelf, index, rule);  
 **return** index;  
 } **else** {  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if** (!rule.getFIRST().contains(**new** Epsilon())) {  
 rule.addToFirst(**new** Epsilon());  
 rule.setChangedFirst(**true**);  
 }**else** {  
 rule.setChangedFirst(**false**);  
 }  
 }  
 **return** index;  
 }  
 }  
  
 // RE1 ::= ('+'|'?'|'\*')? RE  
 **private int** parseRE1(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception{  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*PLUS*) {  
 index++;  
 }  
 **if** (**this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*QUESTION* || **this**.chain.get(index).getTag() == TokenDomainTags.TOKEN\_TAG.*STAR*) {  
 **if** (mode == ParserTags.PARSE\_MODE.*CALCULATE*) {  
 **if**(!rule.getFIRST().contains(**new** Epsilon())) {  
 rule.addToFirst(**new** Epsilon());  
 rule.setChangedFirst(**true**);  
 }**else**{  
 rule.setChangedFirst(**false**);  
 }  
 }  
 index++;  
 }  
 **return** index;  
 }  
  
 // P ::= '(' R ')'  
 **private int** parseP(ParserTags.PARSE\_MODE mode, **int** index, Rule rule) **throws** Exception {  
 index = **this**.parseR(mode, index, rule);  
 **if** (**this**.chain.get(index).getTag() != TokenDomainTags.TOKEN\_TAG.*BR\_CLOSE*) {  
 **throw new** Exception(String.*format*("expected ')' token at position:%s", **this**.chain.get(index).toString()));  
 }  
 index++;  
 **return** index;  
 }  
  
 **private void** parse(ParserTags.PARSE\_MODE mode) **throws** Exception {  
 **if** (chain == **null**) {  
 **throw new** Exception("no tokens");  
 }  
 **try** {  
 **this**.parseS(mode);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }

**public void** calculateFIRST() {  
 **try** {  
 rules = **new** ArrayList<>();  
 rulesMap = **new** HashMap<>();  
 nonterminals = **new** ArrayList<>();  
 **this**.parse(ParserTags.PARSE\_MODE.*INITIAL*);  
 **if** (validateRules()) {  
 **do** {  
 **this**.parse(ParserTags.PARSE\_MODE.*CALCULATE*);  
 } **while** (checkFirstChanged());  
 **this**.printFirsts();  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

Листинг 1. Синтаксический анализатор входного языка методом рекурсивного спуска и вычисление FIRST на языке JAVA.

Результат вычисления FIRST:

../../../var/folders/b2/53l1tl0n2fg8dcsp3109xs840000gn/T/TemporaryItems/NSIRD_screencaptureui_kqL6Y9/Снимок%20экрана%202022-05-06%20

**7. Вывод**

В ходе лабораторной работы был изучен алгоритм построения множеств FIRST для расширенной формы Бэкуса-Наура.