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

Раскрутка самоприменимого компилятора

Вариант 2-Java

Работу выполнил

студент группы ИУ9-61

Бакланова А.Д.

Москва, 2020

Цель работы:

Целью данной работы является изучение алгоритма построения множеств FIRST для расширенной форме Бэкусы-Наура.

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 грамматика.

В качестве *входного языка* должен выступать язык представления правил грамматики, варианты лексики и синтаксиса которого можно восстановить по примерам из таблицы 1.

Индивидуальный вариант:

2	`$NTERM T F E` `$TERM "+" "-" "*" "/"` `$TERM "(" ")" "n"` `$RULE E = T { ("+" | "-") T }` `$RULE T = F { ("*" \| "/") F }` `$RULE F = "n" | "-" F | "(" E ")"`

Задание:

Выполнение данной лабораторной работы состоит из следующих этапов:

1. Составление описаний лексической структуры и грамматики входного языка на основе примера из таблицы 1.
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 не нужно.

В качестве языков реализации разрешается использовать C++, Java, Go, Ruby или Python.

Реализация:

```
199         //Grammar ::= ('$NTERM' Nterms | '$TERM' Terms | '$RULE' Rule)+
200         private static void parse() throws Exception {
201             while (!sym.matches(Tag.END_OF_TEXT)) {
202                 if (sym.matches(Tag.NONTERMS)) {
203                     System.out.println("\nNONTERMS");
204                     sym = sym.next();
205                     do {
206                         parseNonterms();
207                         sym = sym.next();
208 
209                     } while (sym.matches(Tag.NONTERM));
210 
211                 } else if (sym.matches(Tag.TERMS)) {
212                     System.out.println("\nTERMS");
213                     sym = sym.next();
214 
215                     do {
216                         parseTerms();
217                         sym = sym.next();
218                     } while (sym.matches(Tag.TERM));
219                 } else if (sym.matches(Tag.RULE)) {
220                     sym = sym.next();
221                     parseRule();
222                 }
223                 else {
224                     sym.throwError( msg: "error");
225                 }
226             }
227         }
228 
229         //Nterms ::= N+
230         private static void parseNonterms() {
231             System.out.println(sym.start.getWord(sym.follow));
232             grammar.addNonterminal(new Nonterminal(sym.start.getWord(sym.follow)));
233         }
235         //Terms ::= T+
236         private static void parseTerms() {
237             System.out.println(sym.start.getWord(sym.follow));
238             grammar.addTerminal(new Terminal(sym.start.getWord(sym.follow)));
239         }
240 
241         //Rule ::= ( ( N | T | '{' Rule '}' | '(' Rule ')' ) ('|' Rule)?)+
242         private static void parseRule() throws Exception {
243             if (!sym.matches(Tag.NONTERM)) {
244                 sym.throwError( msg: "error");
245             } else {
246                 Nonterminal rule = grammar.getNonterminal(sym.start.getWord(sym.follow));
247                 sym = sym.next();
248                 expect(Tag.ASSIGN);
249                 AST tree = new AST();
250                 do {
251                     tree = parseRule1(tree);
252                 }
253                 while (!sym.matches(Tag.NONTERMS, Tag.TERMS, Tag.END_OF_TEXT, Tag.RULE));
254                 grammar.addRule(rule, tree.node);
255             }
256         }
258          private static AST parseRule1(AST ast) throws Exception {
259              if (sym.matches(Tag.TERM)) {
260                  ast.push(grammar.getTerminal(sym.start.getWord(sym.follow)));
261                  sym = sym.next();
262              } else if (sym.matches(Tag.NONTERM)) {
263                  ast.push(grammar.getNonterminal(sym.start.getWord(sym.follow)));
264                  sym = sym.next();
```

```
        } else if (sym.matches(Tag.LFPAREN)) {
            sym = sym.next();
            AST current = new AST();
            do {
                current = parseRule1(current);
            } while (!sym.matches(Tag.RFPAREN));
            ast.push(new Alt(current.node, new Empty()));
            expect(Tag.RFPAREN);
        } else if (sym.matches(Tag.LPAREN)) {
            sym = sym.next();
            AST current = new AST();
            do {
                current = parseRule1(current);
            } while (!sym.matches(Tag.RPAREN));
            ast.push(current.node);
            expect(Tag.RPAREN);
        } else if (sym.matches(Tag.RPAREN, Tag.RFPAREN)) {
            return ast;
        } else if (!sym.matches(Tag.ALTER)){
            sym.throwError( msg: "error");
        }
        if (sym.matches(Tag.ALTER)) {
            sym = sym.next();
            AST current = parseRule1(new AST());
            ast.node = new Alt(ast.node, current.node);
        }
        return ast;
    }
}
```

Тестирование:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home/bin/java ...

NONTERMS
T
F
E

TERMS
"+"
"-"
"*"
"/"

TERMS
"("
")"
"n"


FIRST:
T: "n" "-" "("
F: "n" "-" "("
E: "n" "-" "("
```

Вывод:

В результате выполнения лабораторной работы был реализован алгоритм построения множеств FIRST для расширенной формы Бэкуса-Нуара.