哈工大软件学院编译原理实验2——语法分析

原创

liushuaikobe 于 2012-11-10 21:49:16 发布 7914 收藏 6
分类专栏： Java 编译原理 文章标签： java Java JAVA 编译原理

这次实验让人煞费苦心啊，话说我已经写了一天的C语言文法了，囧。

总结一下，可以说：程序编写很帅很顺利，文法编写很挫很纠结。我用的是LL(1)分析法（又叫预测分析法），开始的时候花了一段时间来理解LL(1)算法，后来到设计、实现、各种测试，可谓经历了一番波折。记得刚开始写的时候想用C++，后来发现竟然忘的差不多了，囧，于是索性挫到底——用Java实现，轻喷啊。

这次实验的内容就是让你采用一种语法分析技术分析类高级语言中的基本语句，至少包括函数定义、变量说明、赋值、循环、分支等语句，同时还必须让程序有一定的错误处理的机制。

开始要精度ppt和龙书相关的章节，知道这个LL(1)分析法到底是个啥，具体的相关的知识我不再赘述，其实我弄得也不是特别透彻，首先得知道LL(1)分析器的系统结构吧：

有一句话很重要，LL(1)分析器是模拟了最左推导的过程，这句话对理解LL(1)很有帮助。

然后要理解LL(1)的通用控制算法：

漂亮，其实说的通俗一点，LL(1)分析器有一个栈，用来存放推导式，在分析的过程中，由当前栈顶元素和当前输入符号来决定下一步的操作（比如使用哪个产生式继续往下推导，或者弹出栈顶元素前移输入指针，或者报错等操作），另外不要忘了，LL(1)分析法是在模拟最左推导。

根据这个算法，我们要做的事就是构造预测分析表，根据通用控制算法写总控程序，此外，构造预测分析法要用到文法每个产生式的SELECT集，构造SELECT集需要求出每个终结符和非终结符的First集，还要求出每个非终结符的Follow集。

由于代码太多，这次不贴代码了，只是大体说说我的数据结构的设计。

我为非终结符和终结符设计了一个基类（尽管很多情况下我设计的基类被人说成是鸡肋，我还是喜欢这样做）：

```
public abstract class MyCharacter {
 public String what;
 public List<String> First;
 public MyCharacter() {
  First = new ArrayList<String>();
 }
}
```

然后终结符和非终结符继承这个基类：

```
public class TerminateCharacter extends MyCharacter {
 public TerminateCharacter(String what) {
  super();
  this.what = what;
 }
}
```

```
public class NonterminalCharacter extends MyCharacter {
 public List<String> Follow;
 public List<String> Sync;

 public NonterminalCharacter(String what) {
  super();
  this.what = what;
  Follow = new ArrayList<String>();
  Sync = new ArrayList<String>();
 }

 /**
  * 根据一些启发式的方法设置该终结符的同步记号集合，即把该终结符的FOLLOW集、FIRST集加入Sync集
  */
 public void setSync() {
  Sync.addAll(Follow); // 添加Follow集
  for (String s : First) { // 添加First集
   if (!Sync.contains(s)) {
    Sync.add(s);
   }
  }
 }
}
```

此外，产生式也作为程序的一个实体类：

```
public class Production {
 // 产生式的左部
 private String Left;
 // 产生式的右部
 private ArrayList<String> right;
 // 产生式的Select集
 public ArrayList<String> Select;

 public Production(String left, ArrayList<String> right) {
  Left = left;
  this.right = right;
  this.Select = new ArrayList<String>();
 }

 public String getLeft() {
  return Left;
 }

 public void setLeft(String left) {
  Left = left;
 }

 public ArrayList<String> getRight() {
  return right;
 }

 public void setRight(ArrayList<String> right) {
  this.right = right;
 }

}
```

由于语法分析的输入是词法分析的Token序列，因此，我为读到程序里的Token也设计了一个实体类：

```
public class Token {
 private String token;
 private String value;
 public Token(String token, String value) {
  super();
  this.token = token;
  this.value = value;
 }
 public String getToken() {
  return token;
 }
 public void setToken(String token) {
  this.token = token;
 }
 public String getValue() {
  return value;
 }
 public void setValue(String value) {
  this.value = value;
 }
}
```

当一个文法确定下来，这个文法的非终结符、终结符、产生式以及这个文法的预测分析表（前提是该文法必须是LL(1)文法，而且产生式已经消除了回溯和左递归）都会确定下来，他们的数据结构如下：

```
/**
 * 非终结符
 */
public static Map<String, MyCharacter> nCharacters = new HashMap<String, MyCharacter>();
/**
 * 终结符
 */
public static Map<String, MyCharacter> tCharacters = new HashMap<String, MyCharacter>();
/**
 * 产生式
 */
public static List<Production> productions;
/**
 * 预测分析表
 */
public static Map<String, HashMap<String, ArrayList<String>>> ForecastTable = new HashMap<String, HashMap<
```

终结符和非终结符用了一个Map，这个Map用字符本身作为key，该字符同时为这个字符new一个NonterminalCharacter或者TerminateCharacter的对象作为相应的value；预测分析表用了一个二维的Hash表，在查表时可以通过ForecastTable.get(X).get(a)完成。

在写程序的时候要用到一个算法，判断一个非终结符能不能经过N步推导推出空串，这个方法贴出来写下：

```
/**
 * 递归判断X可否推出空串
 *
 * @param X
 * @return
 */
public static boolean canLeadNull(String X) {
 // X是终结符，则X不可能推出空串
 if (isTCharacter(X)) {
  return false;
 }
 // X是非终结符
 else {
  // 查找Cache表
  if (GrammerAnalysis.canLeadNullList.contains(X)) {
   return true;
  }
  for (Production p : GrammerAnalysis.productions) {
   if (X.equals(p.getLeft())) {
    // 存在一个 X=>$ 的产生式，则说明X可以推出空串
    if ("$".equals(p.getRight().get(0))) {
     // 把X加入Cache
     GrammerAnalysis.canLeadNullList.add(X);
     return true;
    }
    // 当前产生式不是 X=>$ ，递归调用
    else {
     boolean flag = true;
     for (int i = 0; i < p.getRight().size(); i++) {
      // 当前产生式不能产生空串
      if (!canLeadNull(p.getRight().get(i))) {
       flag = false;
       break;
      }
     }
     if (flag == true) {
      // 把X加入Cache
      GrammerAnalysis.canLeadNullList.add(X);
      return true;
     }
    }
   }
  }
  return false;
 }
}
```

上面程序的Cache是个啥呢？我们知道，递归一般是很耗资源的，既然文法确定了，那么一个终结符经过判断如果能推出空串，就不会变了，我们把能推出空串的非终结符保存起来，下次直接去查表不就不用再递归判断了？

```
/**
 * 为了提高递归的效率，如果某个终结符经过N步推导后能推出空串，则把这个非终结符放在这个集合中 相当于cache
 */
public static List<String> canLeadNullList = new ArrayList<String>();
```

这个函数的编写是根据一个“公理”，即：一个非终结符序列可以经过N步推导推出空串的充要条件是构成它的每一个非终结符都能经过N步推导推出空串，所以很显然递归是方便快捷的方式。

下面是总控程序：

```
/**
 * 根据当前文法分析句子，输出分析结果
 *
 * @param sentence
 *            要分析的语句（Token表示）
 * @param startChar
 *            当前文法的起始符号
 * @return 返回自顶向下推导序列
 */
public static ArrayList<Production> Analysis(ArrayList<String> sentence,
  String startChar) {
 ArrayList<Production> productionSequences = new ArrayList<Production>();
 Stack<String> prodChars = new Stack<String>();
 prodChars.push("#");
 prodChars.push(startChar);
 // sentence = sentence + "#";
 sentence.add("#");
 int currentIndex = 0; // 当前分析到的下标

 while (!"#".equals(prodChars.peek())) {
  String X = prodChars.peek();
  String a = "";
  if (currentIndex < sentence.size()) {
   a = sentence.get(currentIndex);
  }
  if (isTCharacter(X) || "#".equals(X)) {
   if (a.equals(X)) {
    if (!"#".equals(X)) {
     prodChars.pop();
     currentIndex++;
    }
   } else {
    String eStr = prodChars.pop();
    System.err.println("ERROR,Ignore Char : " + eStr);
    // break;
   }
  } else {
   ArrayList<String> item = GrammerAnalysis.ForecastTable.get(X)
     .get(a);
   if (item != null) {
    prodChars.pop();
    if (!"$".equals(item.get(0))) {
     for (int i = item.size() - 1; i > -1; i--) {
      prodChars.push(item.get(i));
     }
    }
    productionSequences.add(new Production(X, item));
    System.out.println(X + " -> " + item);
   } else {
    if (((NonterminalCharacter) GrammerAnalysis.nCharacters
      .get(X)).Sync.contains(a)) {
     String eStr = prodChars.pop();
     System.err
       .println("ERROR,Have Pop NCharacter: " + eStr);
    } else {
```

```
    } else {
     String eStr = a;
     System.err.println("ERROR,-Ignore Char : " + eStr);
     currentIndex++;
    }
    // break;
   }

  }
 }
 return productionSequences;
}
```

至于C语言文法的编写，我参考了网上流传很广的一份神文法写的。写的各种纠结，各种消除左递归，消除回溯。

由于现在我的文法还有bug，故暂时不贴出来了，代码和数据结构设计仅供参考。

下周有用户界面设计的考试，下下周有计算机安全概论和知识产权法的考试，任重而道远啊。

欢迎留言讨论。

==========

后记：

唉，又调了一下午的文法，终于大部分都基本搞定了，不过这个文法还存在缺陷，还是贴出来吧，毕竟我花了好多心血在里面，各种消除回溯，消除左递归：

缺陷记录：

1.if-else有缺陷，其中selection_statement'的Select集竟然相交了，That is to say，if-else那部分不符合LL（1）文法，即，不支持else子句，求鄙视

2.函数调用只支持带参数的函数调用。postfix_expression' -> (const_expression_list)这一句有点猥琐，如果这句支持无参函数调用的话又会出现if-else的情况。

3.给数组的某一个具体项赋值时有缺陷，不能这样复制b[3] = b[2]。

上述缺陷如果有时间我还愿意去调试，不过有点调不动了，仅供大家参考。

```
program -> external_declaration program'
program' -> external_declaration program' | $
#
external_declaration -> function_definition
#
function_definition -> type_specifier declarator_for_fun compound_statement
#
type_specifier -> CHAR | INT | FLOAT | CHAR*
#
declarator_for_fun -> IDN ( declarator_for_fun'
declarator_for_fun' -> ) | parameter_list )
#
declarator -> IDN declarator'
declarator' -> [ CONST_INT ] declarator' | $
#
#identifer_list -> IDN identifer_list'
```

```
#identifer_list' -> , IDN identifer_list' | $
#
parameter_list -> parameter_declaration parameter_list'
parameter_list' -> , parameter_declaration parameter_list' | $
parameter_declaration -> type_specifier IDN
#
compound_statement -> { compound_statement'
compound_statement' -> } | statement_list } | declaration_list statement_list }
#
declaration_list -> declaration declaration_list'
declaration_list' -> declaration declaration_list' | $
declaration -> type_specifier init_declarator declaration'
declaration' -> , init_declarator declaration' | $
init_declarator -> declarator init_declarator'
init_declarator' -> $ | = initializer
initializer -> assigment_expression | { const_expression_list }
#
statement_list -> statement statement_list'
statement_list' -> statement statement_list' | $
#
statement -> compound_statement | expression_statement | selection_statement | iteration_statement | jump_s
#
expression_statement -> ; | expression ;
#
selection_statement -> IF ( expression ) statement selection_statement'
selection_statement' -> $ | ELSE statement
#
iteration_statement -> WHILE ( expression ) statement | FOR ( expression_statement expression_statement exp
#
jump_statement -> CONTINUE ; | BREAK ; | RETURN ; | RETURN expression ;
#
expression -> assigment_expression expression'
expression' -> , assigment_expression expression' | $
#
assigment_expression -> IDN assigment_expression'' | const_expression assigment_expression'
assigment_expression'' -> assigment_expression' | = logical_or_expression
assigment_expression' -> > logical_or_expression | < logical_or_expression | >= logical_or_expression | <=
logical_or_expression -> logical_and_expression logical_or_expression'
logical_or_expression' -> OR_OP logical_and_expression logical_or_expression' | $
#
logical_and_expression -> equality_expression logical_and_expression'
logical_and_expression' -> AND_OP equality_expression logical_and_expression' | $
#
equality_expression -> relational_expression equality_expression'
equality_expression' -> == relational_expression equality_expression' | $
equality_expression' -> != relational_expression equality_expression' | $
#
relational_expression -> shift_expression relational_expression'
relational_expression' -> > shift_expression relational_expression' | $
relational_expression' -> < shift_expression relational_expression' | $
relational_expression' -> >= shift_expression relational_expression' | $
relational_expression' -> <= shift_expression relational_expression' | $
#
shift_expression -> multiplicative_expression shift_expression'
shift_expression' -> + multiplicative_expression shift_expression' | $
shift_expression' -> - multiplicative_expression shift_expression' | $
#
multiplicative_expression -> cast_expression multiplicative_expression'
multiplicative_expression' -> % cast_expression multiplicative_expression' | $
multiplicative_expression' -> / cast_expression multiplicative_expression' | $
```

```
multiplicative_expression' -> / cast_expression multiplicative_expression' | $
multiplicative_expression' -> * cast_expression multiplicative_expression' | $
#
cast_expression -> postfix_expression
#
postfix_expression -> primary_expression postfix_expression'
postfix_expression' -> [ expression ] postfix_expression' | $
postfix_expression' -> ( const_expression_list )
#
primary_expression -> IDN | const_expression | ( expression )
const_expression -> CONST_INT | CONST_FLOAT | CHAR* | CHAR
#
const_expression_list -> const_expression const_expression_list'
const_expression_list' -> , const_expression const_expression_list' | $
```