哈工大软件学院编译原理实验3——语义分析

原创

liushuaikobe 于 2012-12-08 00:14:01 发布 12091 收藏 8
分类专栏： Java 编译原理 Python 文章标签： java Java JAVA 编译原理

实验目的

这次实验的实验目的其实很明确——对源码进行语义分析，输出语义分析结果，并要求有适当的错误处理机制。可是指导书上实验目的要求自己分析，我的分析结果：本次实验要求自己定义上次实验的语法分析的文法的SDD，然后编写程序在上次语法分析的基础上完成语义分析，生成测试程序的中间代码（三地址码）。

基本概念

本次实验离不开一些概念：语法制导定义（syntax-directed definition (SDD)，我觉得这个翻译略显生硬，“制导”？导弹制导系统？），语法制导翻译（Syntax-Directed Translation（SDT），同上），以及中间代码生成（Intermediate-Code Generation，个人认为把它翻译成“中介”更好一些）。

刚开始可能课上听课程度不够，对这些概念不是很了解，导致做实验时有很大的障碍。

下面说说我的理解：

SDD是一个偏理论上的概念，龙书第二版这样说：

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions（“rules” is “semantic rules”，笔者加）.

也就是说，为了理解语言的含义，我们要把语言符号和语言符号所代表的信息联系起来，我们要为文法的每个grammar symbol(s)附加一些属性。而附加到Production的语义规则则告知这些属性是怎么得来的以及文法符号属性之间的关系是怎样的。我们可以把SDD理解为对单纯的Grammar Symbols和Productions的扩展。SDD让文法的Grammar Symbols和Productions变得“活”了，变得有意义了。

SDT是一种技术（老师的讲义上把它理解为SDD的一种便于书写的形式）。这种翻译技术可以被应用到语义翻译过程中的类型检查和中间代码生成上，也可以被应用到一些某些具有特定任务的轻量级的语言中。可以这么理解，SDT是根据SDD所定义的那些规则和计算顺序对语言进行语义翻译的技术。通过SDT，我们理解了源语言到底要表达个什么意思，为后续的编译工作打下了基础。

中间代码生成，是一个生成一种独立于源语言也独立于目标语言（我们叫它中间代码）的代码过程，中间代码有很多形式，生成中间代码有很多好处，这里不再赘述。中间代码的生成工作要用到SDT技术。

总之，它们都没有什么明确的定义，意会这些就好。

引子：Python实现递归和非递归遍历树

不一定要把SDT作为一个单独的过程去完成它，可能在句法分析的过程中（句法分析当然也有可多方法，LL、LR等）就调用相应的语义子程序完成了SDT的工作。

此外，这次实验我最大的收获：LL(1)语法分析的过程本质上就是一个对语法分析树进行先根遍历的过程。

所以，先写一个简单的脚本实现非递归先根顺序遍历树，来模拟LL（1）的过程，以加深对其的理解。

下面上代码，其中非递归遍历树的过程就是LL(1)执行时遍历语法分析树的过程，读者可以画一下树的结构，并对着输出看一下遍历顺序，会对LL分析过程中分析栈的变化的认识有帮助。

树是这样被录入程序的（tree.txt）：

```
Root:A
A -> B C
B -> D E F
C -> G H
E -> I J
J -> K L M
H -> N O
```

下面是脚本：

```
Tree = {}
Root = None
def read_tree(tree_file_path):
 '''Read the Tree from the file, build the Tree'''
 global Tree,Root
 tree_file = open(tree_file_path,"r+")
 raw_Root = tree_file.readline().strip().split(':')
 if (raw_Root[0] == "Root"):
  Root = raw_Root[1]
 for eachLine in tree_file.readlines():
  sub_tree = eachLine.strip().split(' -> ')
  Tree[sub_tree[0]] = sub_tree[1].split(' ')

def travel_tree(Tree,Root):
 '''Travel the Tree by root-priority order with the stack data structure instead of recursion'''
 stack = []
 stack.append(Root)
 while len(stack) != 0 :
  print 'stack:',stack
  X = stack.pop()
  print 'visit:',X # travel the Tree
  if X not in Tree: # That means X is a leaf
   continue
  for item in list(reversed(Tree[X])):
   stack.append(item)

def travel_tree_recursion(Tree,Root):
 '''Travel the Tree recursively'''
 print Root,
 if Root not in Tree:
  return
 for X in Tree[Root]:
  travel_tree_recursion(Tree,X)

if __name__ == '__main__':
 print "Start reading the tree from file..."
 read_tree('./tree.txt')
 print "The tree is:"
 print Tree,'\n'
 print "Start traveling the tree with stack..."
 travel_tree(Tree,Root)
 print "\nStart traveling the tree recursively..."
 travel_tree_recursion(Tree,Root)
```

输出结果为：

```
Start reading the tree from file...
The tree is:
{'A': ['B', 'C'], 'C': ['G', 'H'], 'B': ['D', 'E', 'F'], 'E': ['I', 'J'], 'H': ['N', 'O'], 'J': ['K', 'L',

Start traveling the tree with stack...
stack: ['A']
visit: A
stack: ['C', 'B']
visit: B
stack: ['C', 'F', 'E', 'D']
visit: D
stack: ['C', 'F', 'E']
visit: E
stack: ['C', 'F', 'J', 'I']
visit: I
stack: ['C', 'F', 'J']
visit: J
stack: ['C', 'F', 'M', 'L', 'K']
visit: K
stack: ['C', 'F', 'M', 'L']
visit: L
stack: ['C', 'F', 'M']
visit: M
stack: ['C', 'F']
visit: F
stack: ['C']
visit: C
stack: ['H', 'G']
visit: G
stack: ['H']
visit: H
stack: ['O', 'N']
visit: N
stack: ['O']
visit: O

Start traveling the tree recursively...
A B D E I J K L M F C G H N O
```

本次实验

下面说一下我的程序的具体实现，仅供参考。

首先要明确，分析栈里面不会再像上次实验那样仅仅存一个文法符号的单一的类型了。这次我们要向栈里面压入至少：文法符号、代表语义动作的“挂钩”、文法符号的属性这些类型。如果想不出更好的将它们类型统一的方法，还是乖乖地把分析栈改成存放Object类型的栈吧：

```
Stack<Object> prodChars = new Stack<>();
```

我们在出栈时，可以利用Java语言的多态性，用instanceof运算符判断弹出来的到底是个什么类型：

```
Object X = prodChars.peek(); // 从栈里弹出元素X
if (X instanceof MyCharacter) { // X是一个文法符号
 // 在这里查找预测分析表，判断该使用哪个产生式
} else if (X instanceof HashMap<?, ?>) { // X是上一个文法符号的综合属性
 // 在这里将综合属性暂存，作为参数传给下一个语义子程序
} else if (X instanceof String) { // X是一个语义子程序
 // 在这里调用相应的语义子程序
}
```

进行SDT的过程，实际上就是前述属性的不断传递的过程，SDD中的属性又分为继承属性（inherited attribute）和综合属性（synthesized attribute）。龙书上是这样描述它们的：

1 . A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

节点N上的综合属性只能通过N的子节点或N本身的属性值来定义。

2. An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

节点N上的继承属性只能通过N的父节点、 N的兄弟节点和N本身的属性值来定义;

根据前面演示脚本的输出，不难得出，按着LL的分析顺序：

1.文法符号的综合属性和由左兄弟节点传递的继承属性是沿着分析栈的栈顶向栈底传递的；
2.其他继承属性是由栈底向栈顶传递的。

为了达到传递两种不同属性的目的，我们需要设计不同的方法：

针对1，由于1的传递顺序是和弹栈顺序相同（栈顶->栈底），则直接在调用相应的语义子程序时顺水推舟传到栈底即可，即在语义子程序中加入诸如：

```
stack[top-n].put(“attribute_name”,”attribute_value”)
```

的操作即可完成栈顶方向属性向栈底传递的操作。

针对2，由于2的传递顺序和弹栈方向相反（栈底->栈顶），也就意味着，对于先入栈的元素，它的属性有可能传递给后入栈的元素，我们可以用全局变量的方式来实现这个顺序的属性传递。先入栈的元素把它的属性放到一个全局变量里面，后入栈的元素根据需要去全局变量里面查找属性即可。

还有一些其他问题，比如属性名重复等等，这个请读者具体问题具体分析自行解决。

做了这些储备之后，就基本可以着手实现了，首先是改造文法，思考并实现SDD，实际上这个过程还是有点难的，可以一步一步地先看看龙书上的样例SDD，再改造自己的文法：

实验二中的文法加入了语义子程序，实际上就是一个字符串，总控程序解析这个字符串判断调用哪个语义动作子程序，下面是我的文法产生式样例：

```
#
type_specifier -> CHAR | INT act2 | FLOAT act3 | CHAR*
#
declarator -> IDN act4 declarator'
declarator' -> [ CONST_INT ] act5 declarator' | $ act6
#
```

总控程序如下：

```
/**
 * 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
 */
@SuppressWarnings("unchecked")
public static ArrayList<Production> Analysis(ArrayList<Token> sentence,
  String startChar) {
 ArrayList<Production> productionSequences = new ArrayList<Production>();
 Stack<Object> prodChars = new Stack<>();
 prodChars.push("#");
 prodChars.push(GrammerAnalysis.nCharacters.get(startChar));
 // sentence = sentence + "#";
 sentence.add(new Token("#", "#"));
 int currentIndex = 0; // 当前分析到的下标
 HashMap<String, String> recentAttr = null; // 语义子程序的参数
 HashMap<String, Stack<String>> attachAttr = new HashMap<>(); // 语义子程序的参数
 attachAttr.put("MostRecentIDN", new Stack<String>());
 attachAttr.put("MostRecentConstInt", new Stack<String>());
 while ((prodChars.peek() instanceof HashMap<?, ?>)
   || (prodChars.peek() instanceof String && !"#".equals(prodChars
     .peek())) || prodChars.peek() instanceof MyCharacter) {
  // outputStack(prodChars);
  Object X = prodChars.peek();
  if (X instanceof MyCharacter) { // X是一个文法符号
   String a = "";
   if (currentIndex < sentence.size()) {
    a = sentence.get(currentIndex).getToken();
   }
   if (X instanceof TerminateCharacter
     || "#".equals(((MyCharacter) X).what)) {
    if (a.equals(((MyCharacter) X).what)) {
     if (!"#".equals(X)) {
      prodChars.pop();
      if ("IDN".equals(((MyCharacter) X).what)) {
       attachAttr.get("MostRecentIDN").push(
         sentence.get(currentIndex).getValue());
      } else if ("CONST_INT"
        .equals(((MyCharacter) X).what)) {
       attachAttr.get("MostRecentConstInt").push(
         sentence.get(currentIndex).getValue());
      }
      currentIndex++;
     }
    } else {
     String eStr = ((MyCharacter) prodChars.pop()).what;
     System.err.println("ERROR,Ignore Char : " + eStr);
     // break;
    }
   } else {
    ArrayList<String> item = GrammerAnalysis.ForecastTable.get(
      ((MyCharacter) X).what).get(a);
    if (item != null) {
     prodChars.pop();
     for (int i = item.size() - 1; i > -1; i--) {
```

```
      if ("$".equals(item.get(i))) {
       continue;
      }
      if (isNCharacter(item.get(i))) {
       prodChars.push(new HashMap<String, String>()); // 非终结符入栈的同时，把用于存放它综合属性的Hash表也入栈
       prodChars.push(GrammerAnalysis.nCharacters
         .get(item.get(i)));
      } else if (isTCharacter(item.get(i))) {
       prodChars.push(GrammerAnalysis.tCharacters
         .get(item.get(i)));
      } else if (item.get(i).startsWith("act")) {
       prodChars.push(item.get(i));
      }
     }
     productionSequences.add(new Production(
       ((MyCharacter) X).what, item));
     System.out.println(((MyCharacter) X).what + " -> "
       + item);
    } else {
     if (((NonterminalCharacter) GrammerAnalysis.nCharacters
       .get(X)).Sync.contains(a)) {
      String eStr = ((MyCharacter) prodChars.pop()).what;
      System.err.println("ERROR,Have Pop NCharacter: "
        + eStr);
     } else {
      String eStr = a;
      System.err.println("ERROR,-Ignore Char : " + eStr);
      currentIndex++;
     }
     // break;
    }
   }
  } else if (X instanceof HashMap<?, ?>) { // X是上一个文法符号的综合属性
   recentAttr = (HashMap<String, String>) prodChars.pop();
  } else if (X instanceof String) { // X是一个语义子程序
   String s = (String) prodChars.pop(); // 解析字符串判断该调用哪个语义子程序
   SDTUtil.execAction(Integer.parseInt(s.substring(3)), prodChars,
     recentAttr, attachAttr);
  }
 }
 return productionSequences;
}
```

在写总控程序时，要时刻牢记LL分析法的语法分析树的遍历的顺序，根据这一点编写总控程序。

此外，Java的Stack类只提供了对当前栈顶元素操作的接口，如果想对非栈顶元素进行操作，可以这样：

```
@SuppressWarnings("unchecked")
public static void act10(Stack<Object> stack,
  HashMap<String, String> recentAttribute,
  HashMap<String, Stack<String>> attachAttribute) {
 Stack<Object> tmpStack = new Stack<>();
 for (int i = 0; i < 2; i++) {
  tmpStack.push(stack.pop());
 }
 ((HashMap<String, String>) stack.peek()).put(
   "postfix_expression.result",
   recentAttribute.get("primary_expression.result"));
 for (int i = 0; i < 2; i++) {
  stack.push(tmpStack.pop());
 }
}
```

上面的语义子程序完成的就是将某一栈顶的属性传递给stack[top - 2]的属性。

这次实验过程有点复杂，但是如果一步一步梳理下来收获还是不小的，虽然可能实验完成的不怎么样，但是也还是很有成就感的，欢迎讨论。

另附上Java中将标准输出重定向到文件流的方法：

```
 try {
  System.setOut(new PrintStream(new FileOutputStream(
    FileAccessUtil.ROOT_DIR + "result.txt")));
 } catch (FileNotFoundException e) {
  e.printStackTrace();
 }
```