编译原理实验PA1-A报告

王力
计63
2016011297

2018 年 10 月 17 日

1 总述

本次实验第一阶段，是完成语法分析器parser能够调用词法分析器lexer，从源代码文本语法结构上生成AST，同时将这个AST打印出来比对正确结果的任务。下面，我分不同语言特性来讲我实现过程中的细节。

2 实现细节

2.1 对象浅复制语句

首先，对于这种新增关键字的，都是先在词法分析器脚本中加入这个关键字，以及对应的识别出这个关键字之后的动作。对于关键字动作来说，就是能够完成把这个“字符串”转化成Parser中能够识别的token的过程。具体来说就是在Parser脚本中使用

这一步完成之后，就是在语法分析器的归约动作中创建AST节点实例的问题。我的做法是，在Tree类中创建一个继承自Tree本身的，叫做Scopy的嵌套类。注意，在创建之前必须要先在Tree中创建对应的tag，这是一个方便检测实例类别的类标识。

创建这个Scopy类来就是作为一个AST节点存储这个关键字语法上应该包含的信息的。类代码在这里贴出，对应Tree中第474行开始。

```
//begin to write new syntax AST Node class enxtends Tree
  public static class Scopy extends Tree {

    public String ident;
    public Expr expr;

      public Scopy(String ident, Expr expr, Location loc) {
          super(SCOPY, loc);
       this.ident = ident;
       this.expr = expr;
      }

    @Override
      public void accept(Visitor v) {
          v.visitScopy(this);
      }

    @Override
    public void printTo(IndentPrintWriter pw) {
       pw.println("scopy");
       pw.incIndent();
       pw.println(ident);
       expr.printTo(pw);
       pw.decIndent();
    }
  }
```

最后，就是在Parser脚本中使用类似上下文无关语言写清归约语法，以及检测到该语法之后创建对应的Scopy实例，然后脚本自动创建自动机完成AST构建。

2.2 禁止类被继承

我的想法是，对于sealed类来说，它本质上并不是和普通类不同，只是特殊的一种类。所以，我同样使用Tree中ClassDef类，但是我在类中加入了一个判断类型的boolean变量sealed，然后修改了构造函数，使得在创建类的

时候显式指明是否可以被继承。

显然，对应parser脚本中，在归约动作中就写一个ClassDef的构造函数，将sealed指明为True就行。代码对应Tree中第558行开始，具体如下。

```
//modified for sealed syntax
 public static class ClassDef extends Tree {

   public boolean sealed;
   public String name;
   public String parent;
   public List<Tree> fields;

     public ClassDef(boolean sealed, String name, String parent,
         List<Tree> fields,
        Location loc) {
      super(CLASSDEF, loc);
      this.name = name;
      this.parent = parent;
      this.fields = fields;
      this.sealed = sealed;
     }

   @Override
     public void accept(Visitor v) {
        v.visitClassDef(this);
     }

   @Override
   public void printTo(IndentPrintWriter pw) {
      pw.println((sealed?"sealed class ":"class ") + name + " "
            + (parent != null ? parent : "<empty>"));
      pw.incIndent();
      for (Tree f : fields) {
         f.printTo(pw);
      }
      pw.decIndent();
   }
}
```

2.3 串行条件卫士语句

对于实现串行条件卫士语句来说，我创建了一个Tree的继承自Tree本身的嵌套类Guard，然后这个类中存储了串行条件卫士语句中，包含两个List，分别存储了所有条件Expr和对应的目标执行语句Stmt（其中Expr必须不为空，但是Stmt可以为空）。

该类对应Tree中第495行开始，代码如下。

```
public static class Guard extends Tree {

   public List<Expr> exprList;
   public List<Tree> stmtList;

   public Guard(List<Expr> exprList, List<Tree> stmtList, Location
       loc) {
      super(GUARD, loc);
      this.exprList = exprList;
      this.stmtList = stmtList;
   }

   @Override
     public void accept(Visitor v) {
         v.visitGuard(this);
     }

   @Override
   public void printTo(IndentPrintWriter pw) {
      pw.println("guarded");
      pw.incIndent();
      int len = exprList.size();
      if (len==0) {
         pw.println("<empty>");
         pw.decIndent();
         return;
      }
      for (int i = 0; i < len; ++i) {
         pw.println("guard");
         pw.incIndent();
```

```
            exprList.get(i).printTo(pw);
            stmtList.get(i).printTo(pw);
            pw.decIndent();
        }
        pw.decIndent();
    }
  }
```

同样还是在Parser脚本中使用上下文无关语法描述归约过程和需要执行的动作。但是值得注意的是，这里有由于有一个递归结构，实际上对应文法中有一个左递归产生式，这里最好使用左递归，因为语法分析器本身对于语义值对象压栈的过程是从左到右的，所以如果采用左递归可以及时从栈顶取出需要的一串结构归约掉。相反，如果采用右递归，很可能溢出分析栈的大小还没有归约。对应parser脚本中的代码如下。

```
GuardedStmt   :  IF '{' GuardMiddleExpr '}'
              {
                 $$.stmt = new Tree.Guard($3.elist, $3.slist, $1.loc);
              }
           ;

GuardMiddleExpr :  IfBranch IfSubStmt
              {
                 $1.elist.add($2.expr);
                 $1.slist.add($2.stmt);
                 $$.elist = $1.elist;
                 $$.slist = $1.slist;
              }
           |  /* empty */
              {
                 $$ = new SemValue();
                 $$.elist = new ArrayList<Tree.Expr>();
                 $$.slist = new ArrayList<Tree>();
              }
           ;

IfBranch   :  IfBranch IfSubStmt GUARDOR
```

```
            {
               $$.elist.add($2.expr);
               $$.slist.add($2.stmt);
            }
          | /* empty */
            {
               $$ = new SemValue();
               $$.elist = new ArrayList<Tree.Expr>();
               $$.slist = new ArrayList<Tree>();
            }
          ;

IfSubStmt    : Expr ':' Stmt
               {
               $$.expr = $1.expr;
               $$.stmt = $3.stmt;
            }
          ;
```

2.4 简单自动类型推导

这里需要注意的是，var x是整体作为左值存在的。所以我在Tree中继承了左值类LValue类，创建了一个叫做Var的嵌套类。这个类中可以存储些类似变量表示符名字，或者类型对象等属性，然后在编译器实验后续阶段中，利用对于赋值表达式右边表达式类型判断来初始化Var中的类型属性。具体代码如下。

```
public static class Var extends LValue {

     public String name;

     public Var(String name, Location loc) {
        super(VAR, loc);
        this.name = name;
        this.loc = loc;
     }

     @Override
```

```
    public void accept(Visitor v) {
       v.visitVar(this);
    }

    @Override
    public void printTo(IndentPrintWriter pw) {
       pw.println("var "+name);
    }
  }
```

2.5 一位数组相关操作

2.5.1 数组常量

由于第一阶段实验中只是输出AST，并不对数组常量中类型是否统一做检查，所以其实是很简单的。由于数组常量也算是一种特殊的字面量，不同的只是它相当于同一类型的字面量的batch，所以我没有创建新的嵌套类。而是直接修改了Tree中的Literal类，在其中加入了常量List。具体代码如下。

```
public static class Literal extends Expr {

    public int typeTag;
      public Object value;

    //for array const syntax
    public List<Expr> constList;

      public Literal(int typeTag, Object value, Location loc) {
          super(LITERAL, loc);
          this.typeTag = typeTag;
          this.value = value;
      }

    //Override construtor for array const syntax
    public Literal(int typeTag, List<Expr> constList, Location loc) {
          super(LITERAL, loc);
          this.typeTag = typeTag;
```

```
        this.constList = constList;
    }

  @Override
    public void accept(Visitor v) {
        v.visitLiteral(this);
    }

  @Override
  public void printTo(IndentPrintWriter pw) {
     switch (typeTag) {
     case INT:
        pw.println("intconst " + value);
        break;
     case BOOL:
        pw.println("boolconst " + value);
        break;
     case ARRAYCONST:
        pw.println("array const");
        pw.incIndent();
        if (constList.size() == 0) {
           pw.println("<empty>");
           pw.decIndent();
           return;
        }
        for (Expr e : constList) {
           e.printTo(pw);
        }
        pw.decIndent();
        break;
     default:
        pw.println("stringconst " + MiscUtils.quote((String)value));
     }
  }
}
```

在实现过程中主要遇到的问题就是，最开始语法设计上存在漏洞。然后在加入了ArrayComp语法之后就发生了移进归约冲突。主要就是因为，

其实这个数组常量语法和后面的数组产生语法都是括号开头的。我在查看了parseroutput之后发现，parser每次会根据未来会读入的一个预测字符来判断移进还是归约，或者使用哪条归约语句。所以就可能产生移进也可以，归约也可以的冲突，然后它缺省选择移进解决冲突，但是这样就使得ArrayComp语法覆盖了ArrayConst语法的检测。

最终，我在parseroutput文件的帮助下，修改了语法，使得能够通过一个预测字符解决冲突。

2.5.2 数组初始化

由于加入了新的二元操作符

这里就不列举源代码了，很少。

2.5.3 数组拼接

和上面一样，拼接语法本质上同样是一个二元操作表达式，但是值得注意的是新加入的操作符需要注明优先级和结合性之类的，包括上面的初始化操作符也需要注明。这样可以避免一些错误归约，以及归约归约冲突。

2.5.4 取子数组表达式

我的思路是，可以将取子数组操作当做是一种特殊的数组index操作，所以可以将方括号中的rangeExpr看做是一种特殊的表达式。而这种rangeExpr表达式又正好是一个二元操作表达式形式，所以可以在归约动作中创建Binary实例即可。没有创建新的类，所以这里不贴代码了。

2.5.5 下标动态访问

这里应该将default语句这个整体看成三元操作符，所以我按照Binary的格式，继承自Binary，创建了一个Ternary三元操作符类，主要区别在于不但存储left、right表达式，还增加存储middle表达式。这里要特别注意default关键字的优先级，结合作业要求，在parser中注明。代码如下。

```
/**
     * A ternary operation.
```

```
    */
  public static class Ternary extends Binary {

    public Expr middle;

    public Ternary(int kind, Expr left, Expr middle, Expr right,
        Location loc) {
      super(kind, left, right, loc);
      this.middle = middle;
    }

    private void ternaryOperatorPrintTo(IndentPrintWriter pw, String
        op) {
      pw.println(op);
      pw.incIndent();
      left.printTo(pw);
      middle.printTo(pw);

      //DIY block, callback function(obj)
      pw.println("default");
      pw.incIndent();
      right.printTo(pw);
      pw.decIndent();

      pw.decIndent();
    }

    @Override
    public void accept(Visitor v) {
      v.visitTernary(this);
    }

    @Override
    public void printTo(IndentPrintWriter pw) {
      switch (tag) {
        case DYNACCESS:
          ternaryOperatorPrintTo(pw, "arrref");
          break;
      }
```

```
    }
  }
```

2.5.6 Python风格数组的表达式

这相当于是一个创建数组的表达式，所以我继承了Expr创建了一个名为ArrayComp的嵌套类，用于存储这个表达式中对应的信息。具体代码如下。

```
public static class ArrayComp extends Expr {

    public Expr generExpr, indexExpr, boolExpr;
    public String name;

    public ArrayComp(Expr generExpr, String name, Expr indexExpr, Expr
        boolExpr, Location loc) {
      super(ARRAYCOMP, loc);
      this.generExpr = generExpr;
      this.name = name;
      this.indexExpr = indexExpr;
      this.boolExpr = boolExpr;
    }

    @Override
    public void accept(Visitor v) {
      v.visitArrayComp(this);
    }

    @Override
    public void printTo(IndentPrintWriter pw) {
      pw.println("array comp");
      pw.incIndent();
      pw.println("varbind "+name);
      indexExpr.printTo(pw);
      if (boolExpr == null) pw.println("boolconst true");
      else boolExpr.printTo(pw);
      generExpr.printTo(pw);
      pw.decIndent();
```

```
    }

  }
```

这个部分值得注意的，就是上述已经提到过的，关于ArrayConst语法一起使用的时候，会产生移进归约冲突的问题。解决方法就是注明优先级或者改进语法本身。

2.5.7 数组迭代foreach语句

本质上来说，这个语法和上面Python风格数组创建语法还是很相似的，主要就是in关键字前面可以使用Var自动类型检测变量或者普通变量。在实现上来说，我同样在Tree中创建了新的嵌套类，代码如下。其他就是对于Var和VarDef两种类型的多态性处理，来实现更加方便的统一形式处理BoundVariable。

```
public static class ForeachLoop extends Tree {

    public Tree boundVariable, loopStmt;
    public Expr indexExpr, boolExpr;

    public ForeachLoop(Tree boundVariable, Expr indexExpr, Expr
        boolExpr, Tree loopStmt, Location loc) {
      super(FOREACH, loc);
      this.boundVariable = boundVariable;
      this.indexExpr = indexExpr;
      this.boolExpr = boolExpr;
      this.loopStmt = loopStmt;
    }

    @Override
    public void accept(Visitor v) {
      v.visitForeach(this);
    }

    @Override
    public void printTo(IndentPrintWriter pw) {
```

```
        pw.println("foreach");
        pw.incIndent();
        if (boundVariable instanceof Var) {
           pw.println("varbind "+((Var)boundVariable).name+" var");
        } else {
           VarDef tmp = (VarDef)boundVariable;
           pw.print("varbind "+tmp.name+" ");
           tmp.type.printTo(pw);
           pw.println();
        }
        indexExpr.printTo(pw);
        if (boolExpr == null) {
           pw.println("boolconst true");
        } else {
           boolExpr.printTo(pw);
        }
        loopStmt.printTo(pw);
        pw.decIndent();
    }
}
```

3 测试结果

这里就不附带图片了，本地给出的样例测试是全部通过的。