编译原理第一次实验报告

**计45 谭思楠 2013011720**

编译原理的第一次实验并没有对于编译器理论本身的特殊要求，主要的意义在于熟悉lex和yacc等编译器构造工具的用法。下面对实验完成的具体思路进行说明：

# 词法分析的改动

## 添加新token

首先，注意到如果要实现题目中要求新添加的语法功能，需要新添加一些token。Token的本身的定义是在Parser.y中定义的。我在Parser.y添加了下列token常量，用于实现新增加的循环和控制语句（使用%token语句进行声明）：

SWITCH CASE DEFAULT CONTINUE REPEAT UNTIL PCLONE

同时加入了下列字符TOKEN，用于三目运算符：

':' '?'

## Lexer.l的修改

为了让词法分析器能够识别新加入的Token，需要将相关的标识符字符串在Lexer.l中进行定义。同时，也需要为三目运算符添加定义规则。使用

TERNARY\_OPERATOR = ("?"|":")

{TERNARY\_OPERATOR} { return operator((int)yycharat(0)); }

定义三目运算符。对于其他的符号，使用类似

"switch" { return keyword(Parser.SWITCH); }

的语法进行定义即可。

## 添加Token的字符串表示

新添加的Token一般要在SemValue类的toString方法中添加新的字符表示。这可以通过在该函数的switch-case中增加一些case分句来实现。

# 语法分析的改动

## 添加运算符（三目和<<）

实现三目运算符和具有特殊意义的”<<” （貌似被叫做PCLONE）运算符的思路较为简单。对于三目运算符首先是要构造一个新的类，下面给出的AST中三目运算符节点的类：

|  |
| --- |
| public static class Ternary extends Expr {  public Expr condition;  public Expr left;  public Expr right;  public Ternary(int kind, Expr condition, Expr left, Expr right, Location loc) {  super(kind, loc);  this.condition = condition;  this.left = left;  this.right = right;  }  @Override  public void accept(Visitor visitor) {  visitor.visitTernary(this);  }  @Override  public void printTo(IndentPrintWriter pw) {  pw.println("conditional-expression");  pw.incIndent();  condition.printTo(pw);  left.printTo(pw);  right.printTo(pw);  pw.decIndent();  }  } |

其中condition表示“？”之前的条件表达式，而“：”两端的表达式分别记为Left和Right。对于每一个AST节点类型，都需要有一个构造函数，这个构造函数用于在.y文件中构造新的节点。PrintTo方法用于实现在输出时打印这个AST节点。此外还有一个当前实现几乎固定的Accept方法。

对应的，在文法定义文件Lexer.l之中，需要添加一些规则在匹配到特定的文法时生成这个Ternary类的对象，具体规则如下：

|  |
| --- |
| Expr : //……  | Expr '?' Expr ':' Expr  {  $$.expr = new Tree.Ternary(Tree.CONDITIONAL, $1.expr, $3.expr, $5.expr, $2.loc);  } | //…… |

同时，需要通过%right语句指定三目运算符的优先级：

%right ':' '?'

对于“<<“运算符的处理是类似的，只不过要简单的多，因为原来的程序中已经有处理二目运算符的类了（Binary），因此只需要添加少量语句和队则，并指定优先级即可。此外，按照要求，需要指定运算符的优先级。因此，三目运算符的结合性声明应当放在最上面，“<<”运算符则应紧随其后。

## 添加switch-case语句

Switch-case语句的实现相对较为复杂，分为以下几个部分：

首先在Parser.y中定义一个CaseStmt非终结符，表示一个Case标签和下面的语句。由于Case标签下面可以有多条语句，所以我使用了代码中之前已经提供的StmtList非终结符来构造Case标签，具体定义如下：

CaseStmt : CASE Constant ':' StmtList

{

$$.stmt = new Tree.Case($2.expr, $4.slist, $1.loc);

}

;

其中Case类的定义思路和上面的类似，包含Literal和List<Tree>类型的成员用于描述其AST的子节点。

Default标签和Case标签的实现是很相似的，唯一的区别就是没有常量表达式。

接下来，需要模仿StmtList非终结符的构造方式构造CaseList，用于处理Case的序列，具体的规则如下：

CaseList : CaseList CaseStmt

{

$$.caselist.add((Tree.Case)$2.stmt);

}

| /\* empty \*/

{

$$ = new SemValue();

$$.caselist = new ArrayList<Tree.Case>();

}

;

为了使上述的定义成与，SemValue类也需要添加一个caselist成员，类型为ArrayList<Tree.Case>。

最后，创造Switch类，这个类有两个构造函数，分别对应有/无Default分句。两者的实质区别在于成员中的Default defaultBlock;究竟会不会被赋值为null。在这个基础上，添加SwitchStmt的语法规则就可以了，最后也要将SwitchStmt加到Stmt的列表里。

## 添加repeat-until语句

Repeat-until语句的添加难度比Switch-Case要小得多，这是因为其结构相对固定，不像Switch-Case一样存在任意多个Case分句，和可选的Default分句。参考While和For循环，给出Repeat-until的语法如下：

RepeatStmt : REPEAT Stmt UNTIL '(' Expr ')'

{

$$.stmt = new Tree.RepeatLoop($2.stmt, $5.expr, $1.loc);

}

;

RepeatLoop类的构造非常简单，使用类似之前的方法即可。

注意到其中只有一个Stmt，因此只有一个Stmt的时候可以没有大括号，否则要加大括号成为一个StmtBlock，StmtBlock本身也是一种特殊的Stmt。最后定义RepeatStmt ';'也是一种Stmt即可。

## 添加continue语句

Continue语句只需要模仿Break语句进行构造即可，添加对应的节点类（只需要一个默认的位置参数），并添加一个非常简单的规则即可。

ContinueStmt : CONTINUE

{

$$.stmt = new Tree.Continue($1.loc);

}

;

# 参考资料

关于repeat-until语句中究竟是应该是用Stmt还是StmtList作为循环体的问题询问了助教。