Decaf_PA1 实验报告

计 31 班 刘智峰 2013011427

【实验一】

1）实验描述：新增++，--运算符，实现自增、自减单操作算子，形如 i++,++i,i--,--i，其语义解释与 C 语言中一致。

2）实现思路：仿照已有的+运算，并观察 TestCases 中的样例，对++、--进行匹配操作，并在 Tree 中的 Unary 类内定义++、--语法分析树的输出。

3）实现说明：

首先，在 lexer.l 中进行++、--的符号匹配：

```
"++"                { return operator(Parser.DOUBLE_PLUS);  }
"--"                { return operator(Parser.DOUBLE_MINUS); }
```

然后，在 Parser.y 中将++、--定义为终结符 DOUBLE_PLUS、DOUBLE_MINUS，规定优先级，并进行对表达式的匹配：

```
%token DOUBLE_PLUS  DOUBLE_MINUS

%right  '?'
%left OR
%left AND
%nonassoc EQUAL NOT_EQUAL
%nonassoc LESS_EQUAL GREATER_EQUAL '<' '>'
%left  '+' '-'
%left  '*' '/' '%'
%nonassoc UMINUS '!'
%nonassoc DOUBLE_PLUS DOUBLE_MINUS
%nonassoc '[' '.'
%nonassoc ')' EMPTY
%nonassoc ELSE
```

```
|   Expr DOUBLE_PLUS
    {
        $$.expr = new Tree.Unary(Tree.POSTINC, $1.expr, $2.loc);
    }
|   DOUBLE_PLUS Expr
    {
        $$.expr = new Tree.Unary(Tree.PREINC, $2.expr, $1.loc);
    }
|   Expr DOUBLE_MINUS
    {
        $$.expr = new Tree.Unary(Tree.POSTDEC, $1.expr, $2.loc);
    }
|   DOUBLE_MINUS Expr
    {
        $$.expr = new Tree.Unary(Tree.PREDEC, $2.expr, $1.loc);
    }
```

由于++、--属于一元运算，我直接在 Unary 类中的输出方法中新加了++、--的输出情况：

```
@Override
public void printTo(IndentPrintWriter pw) {
    switch (tag) {
    case NEG:
        unaryOperatorToString(pw, "neg");
        break;
    case NOT:
        unaryOperatorToString(pw, "not");
        break;
    case POSTINC:
        unaryOperatorToString(pw, "postinc");
        break;
    case PREINC:
        unaryOperatorToString(pw, "preinc");
        break;
    case POSTDEC:
        unaryOperatorToString(pw, "postdec");
        break;
    case PREDEC:
        unaryOperatorToString(pw, "predec");
        break;
    }
```

【实验二】

1) 问题描述：新增三元运算符。实现三操作数算子？：，形如 A？B：C。

2) 实现思路：将“A？B：C”看做一个 Expr，其中 A、B、C 分别为 Expr，因此需编写对应规则 Expr -> Expr ? Expr : Expr

3) 实现说明：

在 lexer.l 中匹配”?”与”:”，因为”?”与”:”均属于简单操作符，所以在 SIMPLE_OPERATOR 中进行添加即可。

在 Parser.y 实现 Expr -> Expr ? Expr : Expr 规则，并在 Tree 定义新的类 QusetionAndColon 来打印输出：

```
Expr '?' Expr ':' Expr
{
    $$.expr = new Tree.QuestionAndColon(Tree.QUESTION_COLON, $1.expr, $3.expr, $5.expr, $1.loc
}
```

```
    * A ?_: operation.
    */
public static class QuestionAndColon extends Expr {

    public Expr left;
    public Expr right;
    public Expr middle;

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

    }

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

    @Override
    public void printTo(IndentPrintWriter pw) {
        pw.println("cond");
        pw.incIndent();
        left.printTo(pw);
        middle.printTo(pw);
        right.printTo(pw);
        pw.decIndent();

    }
}
```

【实验三】

1）问题描述：实现反射运算 numinstances ，形如 numinstances（A）。

其语义解释为：计算结果返回类 A 当前实例对象的个数。

2）实现思路：对照 instanceof()函数来做。通过比较 test2.decaf 的输出和 test_q3.decaf 的输出，发现 numinstances()函数的打印方式和 instanceof()很类似，于是可以根据 instanceof()的实现方式来实现 numinstances()。

3）实现说明：

在 lexer.l 中定义关键字：

```
"instanceof"        { return keyword(Parser.INSTANCEOF);    }
"numinstances"      { return keyword(Parser.NUMINSTANCES);  }
```

在 Parser.y 实现表达式 NUMINSTANCES(IDENTIFIER)：

```
    }
|   NUMINSTANCES '(' IDENTIFIER ')'
    {
        $$.expr = new Tree.NumTest($3.ident, $1.loc);
    }
;
```

在 Tree 中仿照 instanceof 用到的类 TypeTest，实现新的类 NumTest：

```
/**
 * numinstances expression
 */
public static class NumTest extends Expr {

 public String numinstance;

    public NumTest(String instance, Location loc) {
        super(NUMINSTANCES, loc);
     this.numinstance = instance;
    }

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

 @Override
 public void printTo(IndentPrintWriter pw) {
     pw.println("numinstances");
     pw.incIndent();
     pw.println(numinstance);
     pw.decIndent();
 }
}
```

【实验四】

1）问题描述：实现串行条件卫士语句，形如 ***if*** *E*1 : *S*1 ||| *E*2 : *S*2 ||| ... ||| *En* : *Sn* ***fi***。语意见实验说明。

2）实现思路：首先，看到 if，自然参考 if...else 的实现方法。其次，可以发现，中间的 Ei:Si 都是 Expr:Stmt 的类型，于是可以定义一个文法来表示这个类型。然后参照 print 的写法可以发现，print 是输出一个列表中的每一项，与此处需要用到的相同。所以我仿照 print 的写法，将所有 Ei:Si 加到一个列表中，并最终打印这个列表中的每一项即可。

3）实现说明：

在 lexer.l 中定义关键字：

```
"fi"            { return keyword(Parser.FI);          }
```

在 Parser.y 中，参照 ClassList 的写法，将 Expr:Stmt 的形式定义为 GuardedES，并通过 GuardedStmt 来实现 GuardedES 列表，最后定义自己的 GuardedIFStmt 用来打印列表：

```
GuardedES       :   Expr ':' Stmt
                    {
                        $$.guardedES = new Tree.GuardedES($1.expr , $3.stmt , $2.loc);
                    }
                ;

GuardedStmts    :   GuardedES
                    {

                        $$.myList = new ArrayList<Tree.GuardedES>();
                        $$.myList.add($1.guardedES);
                    }
                |
                    GuardedStmts TRIBLE_OR GuardedES
                    {

                        $$.myList.add($3.guardedES);
                    }
                ;

GuardedIFStmt   :   IF GuardedStmts FI
                    {
                        $$.stmt = new Tree.GuardedIFStmt($2.myList , $1.loc);
                    }
                ;
```

其中，用到的列表 myList 在 SemValue.java 文件中定义：

```
// no.4
public List<GuardedES> myList;          public GuardedES guardedES;
```

在 Tree.java 中，参照 VarDef 类定义了 GuardedES 类来维护列表：

```
/**
 * GuardedES
 */
public static class GuardedES extends Tree{

    public Expr expr;
    public Tree tree;
    public GuardedES(Expr _expr , Tree _tree , Location loc){
        super(VARDEF, loc);
        this.expr = _expr;
        this.tree = _tree;
    }
    @Override
    public void accept(Visitor v) {
        v.visitGuardedES(this);
    }
    public void printTo(IndentPrintWriter pw) {
        pw.println("guardedstmt");
        pw.incIndent();
        expr.printTo(pw);
        tree.printTo(pw);
        pw.decIndent();
    }
}
```

同时定义了 GuardedIFStmt 类来打印列表的每一项，得到最后结果：

```
/**
 * no.4 GuardedIFStmt
 */
public static class GuardedIFStmt extends Tree{

    public List<GuardedES> guardedES;

    public GuardedIFStmt(List<GuardedES> _gGuardedES, Location loc) {
        super(GUARDEDIFSTMT, loc);
        this.guardedES = _gGuardedES;
    }

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

    @Override
    public void printTo(IndentPrintWriter pw) {
        pw.println("guardedif");
        pw.incIndent();
        for (GuardedES e : guardedES) {
            e.printTo(pw);
        }
        pw.decIndent();
    }

}
```

【实验五】与实验四只改了首尾关键词 do、od 和打印方式，做法一模一样，此处贴出关键代码：

```
GuaededDOStmt   :   DO GuardedStmts OD
                    {
                        $$.stmt = new Tree.GuardedDOStmt($2.myList , $1.loc);
                    }
                ;
```

```
/**
 * no.5 GuardedDOStmt
 */
public static class GuardedDOStmt extends Tree{

    public List<GuardedES> guardedES;

    public GuardedDOStmt(List<GuardedES> _gGuardedES, Location loc) {
        super(GUARDEDIFSTMT, loc);
        this.guardedES = _gGuardedES;
    }

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

    @Override
    public void printTo(IndentPrintWriter pw) {
        pw.println("guardeddo");
        pw.incIndent();
        for (GuardedES e : guardedES) {
            e.printTo(pw);
        }
        pw.decIndent();
    }

}
```

【实验总结】

通过本次实验的联系，我对 Lexx 和 Yacc 语法有了更深的理解，对语法分析书 AST 也有了一定的认识，为接下来几个阶段的实验打下了良好的基础。

本次实验的难度不算大，主要是刚开始不好上手。感谢计 33 的郭志芃同学给我讲解总体的框架，告诉我去看已经实现的+法的例子，以及多学习利用 TestCases 中的样例。