Decaf PA1_A 实验报告

计64 翁家翌 2016011446

实验一

问题描述

识别形如 `scopy(ident, expr)` 的语句

实现思路

有很多步骤后面是通用的。具体如下：

1. 在 `Lexer.l` 中注册关键字，此处为 `scopy`；
2. 在 `Parser.y` 中注册 `token`，此处为 `SCOPY`；
3. 在 `token` 下方代码设定优先级，此处不必；
4. 写文法，必要的时候在 `Tree.java` 中新建所需变量；
5. 在 `Tree.java` 中新建一个类，用来处理对应文法的语法树，具体为：
    1. 在 300 行之前，注册一个类名大写的变量，此处为 `SCOPY = xxx + 1;`；
    2. 在末尾添加 `visitXXX` 函数，此处为：

    ```
    public void visitScopy(Scopy that) {
        visitTree(that);
    }
    ```

    3. 随便复制一个其他类，重命名为该文法，此处为 `Scopy`；

上述步骤 4 与 5.3 较为关键，以下（可能）会详细讲解。

实现说明

文法实现

```
Stmt -> ScopyStmt | ...
ScopyStmt -> SCOPY '(' IDENTIFIER ',' Expr ')'
```

获取 `ident` 与 `expr` 传给语法树。

语法树实现

由于 `Stmt` 是 `Tree` 类，因此将 `Scopy` 类继承 `Tree`。

……感觉十分简单，是拿来熟悉代码用的，就不讲了吧（

实验二

问题描述

`class` 之前增加关键字 `sealed`

实现思路

本来是想写个类来判断 `sealed`，但是始终不行，会报 `Error`。经同学指点，发现只要把 `ClassDef` 稍微修改一下即可。

实现说明

文法实现

将 `ClassDef` 修改为

```
ClassDef -> CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
  |  SEALED CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
```

也就是添加了对于 `sealed` 的特殊处理。同时修改语法树接口，指明该类是否被 `sealed` 修饰。

语法树实现

将 `ClassDef` 的构造函数修改为

```
public ClassDef(boolean sealed, String name, String parent, List<Tree> fields,
Location loc) { ... }
```

参数 `sealed` 表示是否被 `sealed` 修饰，并且根据该参数修改相应输出。

实验三

问题描述

串行条件卫士：`if { E1 : S1 ||| E2 : S2 ||| ... ||| En : Sn }`。`n` 可以为 `0`。

实现思路

在 `Lexer.l` 中添加 `|||` 操作符识别，`SIMPLE_OPERATOR` 中添加对 `:` 的正则匹配。

之后将其分为一个个小的模块，称之为 `GuardedES`，里面包含 `Ex : Sx`。由于这玩意可以有很多，因此还要在 `SemValue.java` 里面开个 `List` 进行记录。

实现说明

文法实现

```
Stmt -> GuardStmt | ...
GuardStmt -> IF '{' GList '}' | IF '{' '}' # 此处判掉空
GList -> GList TRI_OR GuardedES | GuardedES
GuardedES -> Expr ':' Stmt
```

同时，`GList` 中维护变量 `glist`，用 `List` 来存储 `GuardedES`。

语法树实现

创建两个类 `GuardedES` 和 `GuardStmt`。

类 `GuardedES` 的构造函数为

```
public GuardedES(Expr expr, Tree stmt, Location loc) { ... }
```

类 `GuardStmt` 的构造函数为

```
public GuardStmt(List<GuardedES> guardedES, Location loc) { ... }
```

如果 `guardedES` 为空，那么输出 `<empty>`。

实验四

问题描述

识别 `var ident`

实现思路

根据提示，直接在 `LValue` 中进行修改。

实现说明

文法实现

将 `LValue` 的文法改为：

```
LValue -> VAR IDENTIFIER | ...
```

语法树实现

新建类 `VarIdent`，继承自 `LValue`。步骤较简单。

实验五

（1）

问题描述

支持数组常量，形如 `[1, 2, 3]`

实现思路

拓展 `Constant` 的文法，利用 `elist` 记录各个常量并传给语法树。

实现说明

文法实现

```
Constant -> '[' ']' | '[' ConstList ']' # 此处判掉空
ConstList -> ConstList ',' Constant | Constant
```

并且使用 `elist` 记录 `Constant`。

语法树实现

新建类 `ArrayConst`，构造函数为：

```
public ArrayConst(List<Expr> constlist, Location loc) { ... }
```

接受一个 `List`。如果其长度为 `0` 则说明为空。

（2）

问题描述

识别形如 `Expr %% intConst` 的语句

实现思路

直接在 `Expr` 中的文法改。然而按照readme直接做会wa，必须得写成 `Expr %% Expr` 才行……

实现说明

文法实现

由于它是左结合，在 `Parser.y` 中加入 `left MOMO` ；由于优先级设定，将其插入至 `%left '+' '-'` 上方。修改 `Expr` 的文法为：

```
Expr -> Expr MOMO Expr | ...
```

语法树实现

在原来的 `Binary` 类中直接修改，加入 `case MOMO` 的输出即可。

（3）

问题描述

识别 `Expr ++ Expr`

实现思路

和上面很类似，直接照着做一遍。

实现说明

文法实现

由于它是右结合，在 `Parser.y` 中加入 `right PLPL` ；由于优先级设定，将其插入至 `%left MOMO` 上方。修改 `Expr` 的文法为：

```
Expr -> Expr PLPL Expr | ...
```

语法树实现

在原来的 `Binary` 类中直接修改，加入 `case PLPL` 的输出即可。

（4）

问题描述

实现数组分片的识别：`Expr[Expr : Expr]`

实现思路

同上

实现说明

文法实现

修改 `Expr` 的文法为

```
Expr -> Expr '[' Expr ':' Expr ']' | ...
```

语法树实现

新建类 `SubArray`，其构造函数为

```
public SubArray(Expr array, Expr begin, Expr end, Location loc) { ... }
```

输出很简单。

（5）

问题描述

识别 `Expr [ Expr ] default Expr`

实现思路

添加关键字 `default`，设置优先级。

实现说明

文法实现

由于 `dafault` 优先级设置，将代码做如下修改：

```
%right PLPL
%left MOMO
%left  '+' '-'
%left  '*' '/' '%'
%nonassoc UMINUS '!'

->

%right PLPL
%left MOMO
%left  '+' '-'
%left  '*' '/' '%'
%nonassoc UMINUS '!' DEFAULT
```

并且修改 `Expr` 的文法为

```
Expr -> Expr '[' Expr ']' DEFAULT Expr | ...
```

语法树实现

新建类 `Default`，其构造函数为

```
public Default(Expr array, Expr index, Expr deft, Location loc) { ... }
```

输出很简单。

（6）

问题描述

识别Python风格数组：`[ Expr for ident in Expr <if BoolExpr> ]`

实现思路

将是否含有 `if` 的情况分类讨论。其他照旧。

实现说明

文法实现

修改 `Expr` 的文法为

```
Expr -> '[' Expr FOR IDENTIFIER IN Expr ']'
     |  '[' Expr FOR IDENTIFIER IN Expr IF Expr']'
```

并且针对第一种情况，往语法树传一个 `true`，具体为 `new Tree.Literal(Tree.BOOL, true, $7.loc)`。虽然可以hardcode输出，但是万一后面PA改需求了呢（

语法树实现

新建类 `PyArray`，其构造函数为

```
public PyArray(Expr expr1, String ident, Expr expr2, Expr expr3, Location loc) { ...
}
```

由于直接传了个封装好的 `true` 进来，因此不必特判输出，直接调用相应接口即可。

（7）

问题描述

识别形如 `foreach (var ident in Expr <while Expr>) Stmt` 或者把 `var` 改成 `Type` 的语句

实现思路

`while` 的条件同(6)处理。

由于输出需要，但是 `var` 又不能直接放进 `Type` 中，因此新开一个类 `ExDef` 来专门处理 `var + Type` 的东西。这样可以避免传统方法枚举的低效。

实现说明

文法实现

```
Stmt -> ForEachStmt | ...
ForEachStmt -> FOREACH '(' BoundVariable IN Expr ')' Stmt
            |  FOREACH '(' BoundVariable IN Expr WHILE Expr ')' Stmt
BoundVariable -> Type IDENTIFIER | VAR IDENTIFIER
```

并且在 `BoundVariable` 中维护变量 `exdef`，处理 `var + Type` 的情况。

语法树实现

新建类 `Exdef` 和 `ForEach`，其中 `ExDef` 的构造函数为：

```
public ExDef(String name, TypeLiteral type, boolean isvar, Location loc) { ... }
```

如果为 `Type` 则 `type` 非空并且 `isvar == false`；如果是 `var` 则 `type == null` 并且 `isvar == true`。

`ForEach` 的构造函数为：

```
public ForEach(ExDef exdef, Expr expr1, Tree stmt, Expr expr2, Location loc) { ... }
```

输出的时候，`foreach` 中从 `varbind` 之后一直到回车的地方交给 `ExDef` 输出。

实验总结

通过本次实验的练习，我对 Lex 和 Yacc 语法有了更深的理解，对 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，在此十分感谢计 63 的刘家硕同学给我讲解总体的框架和实现要领。