Decaf PA1-A 实验报告

2017011462 方言

1、特性一：抽象类

加入`abstract`关键字，可用于修饰类和成员函数。

实现思路和步骤

- 在`Decaf.jacc`中
  - 加入此token：`%token ABSTRACT`
  - 在`ClassDef`中加入新的语法规范

```
ABSTRACT CLASS Id ExtendsClause '{' FieldList '}'
{
$$ = svClass(new ClassDef(true, $3.id, Optional.ofNullable($4.id), $6.fieldList,
$2.pos));
}
```

  - 同理在`MethodDef`中也加入新的语法规范，注意到此产生式中没有`Block`，因此传入`Optional.empty()`

```
ABSTRACT Type Id '(' VarList ')' ';'
{
$$ = svField(new MethodDef(false,true,$3.id,$2.type,
$5.varList,Optional.empty(),$3.pos));
}
```

- 在`Decaf.jflex`中，加入此关键字：`"abstract" { return keyword(Tokens.ABSTRACT); }`
- 在`Tree.java`中，具体实现
  - 在`ClassDef`类中，修改其构造函数，增加参数`isAbstract`

```
public ClassDef(boolean isAbstract, Id id, Optional<Id> parent, List<Field>
fields, Pos pos)
{
super(Kind.CLASS_DEF, "ClassDef", pos);
this.id = id;
this.parent = parent;
this.fields = fields;
this.name = id.name;
this.modifiers = isAbstract ? new Modifiers(Modifiers.ABSTRACT, pos) : new
Modifiers();
}
```

这里参考了`MethodDef`中的实现方法，增加了成员变量`modifiers`，用来记录是否是`Abstract`信息。

当然，在成员函数`treeElementAt`和`treeArity`中都做对应修改，增加此`modifiers`变量。

- 在`MethodDef`类中，修改其构造函数，增加参数`isAbstract`

  具体实现与`ClassDef`类似，参照了`Static`的实现方式，主要修改其`modifiers`

  ```
  this.modifiers = isStatic ? new Modifiers(Modifiers.STATIC, pos) : (isAbstract ?
  new Modifiers(Modifiers.ABSTRACT, pos) : new Modifiers());
  ```

- 在`Modifiers`类中，增加`ABSTRACT`，这里仿照`STATIC`实现

  ```
  public static final int STATIC = 1;
  public static final int ABSTRACT = 2;
  ```

  并且在构造函数中，添加对应的输出`flag`

  ```
  if (isStatic()) flags.add("STATIC");
  if (isAbstract()) flags.add("ABSTRACT");
  ```

- 在`Tokens.java`中加入此关键字：`int ABSTRACT = 31;`

由此即可完成特性一，难度并不是很大，主要用于熟悉框架，比较困难的点是`Optional.empty()`，在原有的产生式中并没有直接出现，因此花了一些时间。

2、特性二：局部类型推断

加入`var`关键字，用来修饰局部变量

实现思路和步骤

- 与特性一中加入`abstract`关键字的做法类似，在几个特定文件中加入`VAR`，此处不多叙述。
- 在`Decaf.jacc`中，在`SimpleStmt`中加入新的语法规范：

  ```
  VAR Id '=' Expr
  {
  $$ = svStmt(new LocalVarDef(Optional.empty(), $2.id, $2.pos,
  Optional.ofNullable($4.expr), $2.pos));
  }
  ```

- 在`Tree.java`的`LocalVarDef`类中：
  - 由于此种语法规范不需要指定`TypeLit`，并且在标准输出中此处对应输出为`<none>`。参考对比一下其他部分可知，应当将`typeLit`改为`Optional`的变量：

    ```
    public Optional<TypeLit> typeLit;
    ```

  - 与之对应地去修改构造函数

```
public LocalVarDef(Optional<TypeLit> typeLit, Id id, Pos assignPos, Optional<Expr>
initVal, Pos pos)
```

- 在之前`Decaf.jacc`中调用此构造函数时，传入`Optional.empty()`

由此即可实现特性二，具体流程和难度都与特性一差不多，要注意将使用`LocalVarDef`的地方都改为`Optional`变量的问题。

3、特性三：First-class Functions

3.1 函数类型

- 在`Decaf.jacc`中：
  - 在`Type`中，增加新的语法规范
  - 增加新的`TypeList`（和`TypeList1`），这部分实现参考`VarList`（和`VarList1`）

    `TypeList`中由一个或多个`Type`，以`,`隔开组成，或者可以为空。这与`VarList`的定义类似，因此仿照其实现即可。
- 在`Tree.java`中：
  - 增加新的类`TLambda`，这部分实现参考其他实现（`TInt`，`TClass`等）

    `TLambda`类中，需要记录返回的类型`returnType`，以及一个`typeList`

    ```
    public static class TLambda extends TypeLit {
        public TypeLit returnType;
        public List<TypeLit> typeList;
        public TLambda(TypeLit type, List<TypeLit> typeList, Pos pos)
        {
            super(Kind.T_LAMBDA, "TLambda", pos);
            this.returnType = type;
            this.typeList = typeList;
        }
    }
    ```
  - 增加新的`Kind:TLambda`
- 在`SemValue.java`中：
  - 增加新的成员变量`List<Tree.TypeLit> typeList;`
- 在`AbstractParser.java`中：增加函数`SemValue svTypes(...)`，这部分实现参照`svStmt`和`svStmts`

```
protected SemValue svTypes(Tree.TypeLit... types)
{
    var v = new SemValue(SemValue.Kind.TYPE_LIST, types.length == 0 ? Pos.NoPos :
types[0].pos);
    v.typeList = new ArrayList<>();
    v.typeList.addAll(Arrays.asList(types));
    return v;
}
```

至此可以实现函数类型。

3.2 Lambda表达式

实现流程基本类似，此处做简要叙述。较为关键的部分为：

- 注册新的关键字`FUN`和操作符`ARROW`，分别对应`fun`和`=>`，这里不再详细叙述。同时设置`=>`的优先级为最低。
- 增加新的语法规范（产生式）：

```
FUN '(' VarList ')' ARROW Expr
{
$$ = svExpr(new Lambda(false, $3.varList, Optional.ofNullable($6.expr),
Optional.empty(), $1.pos));
}

|

FUN '(' VarList ')' Block
{
$$ = svExpr(new Lambda(true, $3.varList, Optional.empty(),
Optional.ofNullable($5.block), $1.pos));
}
```

  此处参数列表实现参考`MethodDef`，采用`VarList`

- 增加对应的类`Lambda`，需要在记录是否为`Block`形式，增加对应的`Kind:Lambda`

```
public static class Lambda extends Expr {
    public List<LocalVarDef> params;
    public Optional<Block> body;
    public Optional<Expr> expr;
    public boolean isBlock;
    public Lambda(...)
    {
        super(Kind.LAMBDA_EXPR, "Lambda", pos);
        this.params = params;
        this.expr = expr;
        this.body = body;
        this.isBlock = isBlock;
    }
    ...
}
```

3.3 函数调用

- 在Decaf.jacc中直接修改产生式：

```
Expr '(' ExprList ')'
{
    $$ = svExpr(new Call(Optional.ofNullable($1.expr), $3.exprList, $2.pos));
}
```

- 在Tree.java中修改类`Call`：

```
public static class Call extends Expr
{
    public Optional<Expr> expr;
    public List<Expr> args;
    public Call(Optional<Expr> expr, List<Expr> args, Pos pos)
    {
        super(Kind.CALL, "Call", pos);
        this.expr = expr;
        this.args = args;
    }
}
```

  去掉成员变量`method`和`id`，同时修改构造函数等。

  这部分遇到了一些问题。最初我的产生式使用了原有的`Receiver`，但是在有一个测例中，`Call`和`ADD`操作的优先级顺序出了问题，后来发现`Receiver`可能生成空，所以将其换成了`Expr`，这样解决了问题。

由此即可完成特性三的实现。

3. 问题

Q1：AST 结点间是有继承关系的。若结点 `A` 继承了 `B`，那么语法上会不会 `A` 和 `B` 有什么关系？限用 100 字符内一句话说明。

答：若A继承B，则语法上存在形如`B->A`的产生式，或者由B可以推出A。

Q2：原有框架是如何解决空悬 else (dangling-else) 问题的？限用 100 字符内说明。

答：原有框架中设定了empty的优先级低于else，因此是优先匹配else语句，由此可知是else语句优先与最后出现的未匹配的if语句相匹配。

Q3：输入程序 lex 完得到一个终结符序列，然后构建出具体语法树，最后从具体语法树构建抽象语法树。 这个概念模型与框架的实现有什么区别？我们的具体语法树在哪里？限用 120 字符内说明。

答：框架中的词法分析和文法分析并不是两个分开的阶段，而是按需解析，在文法分析去需要时才获取下一个单词，并且直接由单词流构造AST。具体语法树没有直接构造，但在`Decaf.jacc`中，把用到的产生式用树状结构展开即可得到具体语法树。