编译原理 PA1-A 报告

计 71 张程远 2017011429

1 工作简介

本次作业要求在原有的 decaf 编译框架里增加 3 个新的语言特性，下面是我实现这些特性的简要过程。

1.1 抽象类与抽象方法

（1）首先在 jflex、jacc 以及 tokens、jacctokens 中增加这一 token，使得程序认识这个关键词。

（2）之后按照实验指导在 jacc 中增加语法。**为了增加鲁棒性，这里尽量不改变原有的构造函数，而是根据需求在原本的类中增加新的构造函数。接下来的实现也基本遵守这一原则。**

```
1.  ABSTRACT CLASS Id ExtendsClause '{' FieldList '}'
2.            {
3.                 $$ = svClass(new ClassDef(true, $3.id, Optional.ofNullable($4
    .id), $6.fieldList, $2.pos));
4.            }
5.        ;
```

```
1.  ABSTRACT Type Id '(' VarList ')' ';'
2.             {
3.                 $$ = svField(new MethodDef($3.id, $2.type, $5.varList, $3.pos
    ));
4.             }
5.         ;
```

（3）接下来为这一语法实现语法树的逻辑，这一部分可以模仿 static 关键字的实现。首先输出需要我们打印“ABSTRACT”，碰巧 static 也有类似的要求，于是在 modifier 中增加了 abstract 的相应内容，增加时照抄 static 的部分就可以了；其次 tree 需要判断 jacc 是否选择 abstract，于是在原来的构造函数中增加了新的 boolean 参数。其他部分模仿给出的函数。

ClassDef 部分：

```
1.          public ClassDef(boolean isAbstract, Id id, Optional<Id> parent, List
    <Field> fields, Pos pos) {
2.              super(Kind.CLASS_DEF, "ClassDef", pos);
3.              this.modifiers = new Modifiers(Modifiers.ABSTRACT, pos);
4.              this.id = id;
5.              this.parent = parent;
6.              this.fields = fields;
7.              this.name = id.name;
8.          }
```

MethodDef 部分：

```
1.  public MethodDef(Id id, TypeLit returnType, List<LocalVarDef> params, Pos po
    s) {
2.              super(Kind.METHOD_DEF, "MethodDef", pos);
3.              this.modifiers = new Modifiers(Modifiers.ABSTRACT, pos);
4.              this.id = id;
5.              this.returnType = returnType;
6.              this.params = params;
7.              this.name = id.name;
8.          }
```

为了应对 body 是否存在的问题，加入了判断是否是 abstract 的函数，并将 case4 改成了下面的写法：

```
case 4 -> isAbstract()? Optional.empty() : body;
```

至此完成 abstract 关键词的添加。

1.2 var 类型变量

（1）首先注册变量，与 1.1 的第一步相同。

（2）增加语法，直接照搬实验指导书的提示。

```
1.  Stmt:  ...  |   VAR Id '=' Expr
2.              {
```

```
3.                  $$ = svStmt(new LocalVarDef($2.id, $3.pos, $4.expr, $2.pos))
    ;
4.              }
5.          ;
```

（3）实现语法树的逻辑。观察输出可知，var 修饰的变量都属于 LocalVarDef 的部分，只是不需要类型的指定，于是在 LocalVarDef 里提供新的构造函数即可。

LocalVarDef ： 加入 boolean isvar 判断是不是 VAR 类型，并加入新的构造函数。

```
1.  public LocalVarDef(Id id, Pos assignPos, Expr initVal, Pos pos){
2.              super(Kind.LOCAL_VAR_DEF, "LocalVarDef", pos);
3.              this.isvar = true;
4.              this.id = id;
5.              this.assignPos = assignPos;
6.              this.varinit = initVal;
7.              this.name = id.name;
8.          }
```

在其他构造函数里将 isvar 设置为 false。之后在遍历 element 时修改代码为：

```
1.  case 0 -> isvar? Optional.empty() : typeLit;
2.  case 1 -> id;
3.  case 2 -> isvar? varinit : initVal;
```

这里由于 var 类型必须要有初始值，所以为 var 提供了单独的初始语段变量。

至此完成 var 的添加。

1.3 Lambda 表达式

1.3.1 函数类型

（1）根据提供的正则在 jacc 中实现语法。这里的 TypeList 是模仿 VarList 写的。

```
1.  Type ...            |   Type '(' TypeList ')'
2.                      {
3.                          $$ = svType(new TLambda($1.type, $3.typelist, $1.typ
    e.pos));
4.                      }
```

```
                ;
TypeList        :   TypeList1
                    {
                        $$ = $1;
                    }
                |   /* empty */
                    {
                        $$ = svTypes();
                    }
                ;
TypeList1       :   TypeList1 ',' Type
                    {
                        $$ = $1;
                        $$.typelist.add($3.type);
                    }
                |   Type
                    {
                        $$ = svTypes($1.type);
                    }
                ;
```

这里还需要模仿 svVars()实现 svTypes()，在 AbstractParser 里。

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

接下来实现节点 TLambda。它属于 Type，因此继承 TypeLit；这里是新加的节点，需要在 Visitor 里增加新的函数 visitTLambda()。其他工作主要是根据输出来确定节点内的元素。

```
public static class Tlamb（）da extends TypeLit{
        public TypeLit typeLit;
        public List<TypeLit> types;

        public TLambda(TypeLit returnType, List<TypeLit> types, Pos pos){
            super(Kind.T_LAMBDA,"TLambda",pos);
            this.typeLit = returnType;
            this.types = types;
        }
        @Override
        public Object treeElementAt(int index){
            return switch (index) {
                case 0 -> typeLit;
                case 1 -> types;
                default -> throw new IndexOutOfBoundsException(index);
            };
        }
        @Override
        public int treeArity(){return 2;}

        public <C> void accept(Visitor<C> v, C ctx){v.visitTLambda(this, ctx
);}
    }
```

1.3.2 Lambda 表达式

（1）首先注册 fun 这一 token。

（2）根据实验指导的正则来写语法。这里运算符'=>'在语法树中根本没有输出，但仍然需要注册，并且由于它是两个字符组成的，因此在 jflex 中我将其定义为 Tokens.DERIVATE，然后在 jacc 注册 DERIVATE 这个关键词，并将其写在所有运算符的最上方（优先级最低），结合方式为%right。语法定义如下。

```
|   FUN '(' VarList ')' Block
```

```
    {
        $$ = svExpr(new Lambda($3.varList, $5.block, $1.pos));
    }
|   FUN '(' VarList ')' DERIVATE Expr
    {
        $$ = svExpr(new Lambda($3.varList, $6.expr, $1.pos));
    }
;
```

（3）接下来去 tree 中实现它。Ifdev 代表它是否有'=>'符号，由此判断它是否有 block。

```
public static class Lambda extends Expr {
    public List<LocalVarDef> params;
    public Block body;
    public Expr obj;
    public boolean ifdev;

    public Lambda(List<LocalVarDef> params, Block body, Pos pos){
        super(Kind.LAMBDA, "Lambda", pos);
        this.params = params;
        this.body = body;
        this.ifdev = false;
    }
    public Lambda(List<LocalVarDef> params, Expr expr, Pos pos){
        super(Kind.LAMBDA, "Lambda", pos);
        this.params = params;
        this.obj = expr;
        this.ifdev = true;
    }
    @Override
    public Object treeElementAt(int index) {
        return switch (index) {
```

```
            case 0 -> params;
            case 1 -> ifdev? obj : body;
            default -> throw new IndexOutOfBoundsException(index);
        };
    }

    @Override
    public int treeArity() {
        return 2;
    }

    public <C> void accept(Visitor<C> v, C ctx) {
        v.visitLambda(this, ctx);
    }
}
```

1.3.3 Call 函数

（1）直接修改 jacc 中的内容。这里我们无法再贯彻“不动原有构造函数“的原则，因为这里指出要修改为要求的形式。直接按实验指导进行修改即可。找到调用 call 的部分直接修改。

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

（2）在 Tree 中直接修改。代码如下所示：

```
  public static class Call extends Expr {
        // Tree elements
//      public Id method;
        public Expr expr;
        public List<Expr> args;
        //

```

```
8.          public Call(Expr expr, List<Expr> args, Pos pos) {
9.              super(Kind.CALL, "Call", pos);
10.             this.expr = expr;
11.             this.args = args;
12.         }
13. /*
14.         public Call(Id method, List<Expr> args, Pos pos) {
15.             this(Optional.empty(), method, args, pos);
16.         }
17.
18.         public Call(Expr receiver, Id method, List<Expr> args, Pos pos) {
19.             this(Optional.of(receiver), method, args, pos);
20.         }
21. */
22.         /**
23.          * Set its receiver as {@code this}.
24.          * <p>
25.          * Reversed for type check.
26.          */
27. /*        public void setThis() {
28.             this.receiver = Optional.of(new This(pos));
29.         }
30. */
31.         @Override
32.         public Object treeElementAt(int index) {
33.             return switch (index) {
34. //                case 0 -> receiver;
35. //                case 1 -> method;
36. //                case 2 -> args;
37.                 case 0 -> expr;
```

```
                case 1 -> args;
                default -> throw new IndexOutOfBoundsException(index);
            };
        }

        @Override
        public int treeArity() {
            return 2;
        }

        @Override
        public <C> void accept(Visitor<C> v, C ctx) {
            v.visitCall(this, ctx);
        }
    }
```

至此第三部分特性修改完成。

2 思考题

(1) 语法上 A 是 B 的一种具体实现，如 Tbool 是其中一种 TypeLit，正如 bool 是一种 type。

(2) 只有在 if 后面才会出现 ElseClause 部分，所以 else 必须跟随 if 出现。

(3) CST 是由 jacc 产生的，它体现在 DecafJaccParser 的 yysv 数组中，因此产生 CST 与 AST 并无时间上的先后之分，相当于框架将两个阶段合二为一了。这是框架与流程不同的地方。