PA1-A实验报告

计73 王焱 2017050024

一、实验综述

本次实验主要是在已有的框架上加入三个新的语法特性，实现 Decaf 语言编译器的词法分析和语法分析部分，同时生成抽象语法树。

二、具体实现

本次实验一共需要实现三个新特性，分别为 `abstract` 、`var`、`fun` 。

首先需要在`Decaf.jflex`中需要定义几种`Tokens`并在`Tokens.java`中为其添加整数值，以及在`JaccParser.java`和`Decaf.Jacc`中进行注册。

```
"abstract"          { return keyword(Tokens.ABSTRACT);     }
"var"               { return keyword(Tokens.VAR);          }
"fun"               { return keyword(Tokens.FUN);          }

    // operators, with more than one character
"=>"                { return operator(Tokens.LAMBDADEF);     }
```

```
 int ABSTRACT = 31;
 int VAR = 32;
 int FUN = 33;
 int LAMBDADEF = 34;
```

```
  case Tokens.ABSTRACT -> decaf.frontend.parsing.JaccTokens.ABSTRACT;
  case Tokens.VAR -> decaf.frontend.parsing.JaccTokens.VAR;
  case Tokens.FUN -> decaf.frontend.parsing.JaccTokens.FUN;
  case Tokens.LAMBDADEF -> decaf.frontend.parsing.JaccTokens.LAMBDADEF;
```

```
%token ABSTRACT     VAR     FUN    LAMBDADEF
```

1.抽象类：加入 abstract 关键字，用来修饰类和成员函数。

首先在 Decaf.jacc 中添加相应的文法。

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

```
MethodDef       :   STATIC Type Id '(' VarList ')' Block
                    {
                        $$ = svField(new MethodDef(false, true, $3.id, $2.type,
$5.varList, $7.block, $3.pos));
                    }
                |   ABSTRACT Type Id '(' VarList ')'  ';'
                    {
                        $$ = svField(new MethodDef(true, false, $3.id, $2.type,
$5.varList, null, $3.pos));
                    }
                |   Type Id '(' VarList ')' Block
                    {
                        $$ = svField(new MethodDef(false, false, $2.id, $1.type,
$4.varList, $6.block, $2.pos));
                    }
                ;
```

修改`tree.java`中相应的`ClassDef`和`MethodDef`类。在`ClassDef`中添加`Modifiers`成员变量，并相应的修改构造函数以及`treeElementAt`和`treeArity`函数。在`MethodDef`中对于不同的修饰如`static`或`abstract`构造不同的`Modifiers`。另外需要修改`Modifiers`类，添加对于`abstract`属性的支持。

```
public static class ClassDef extends TreeNode {
        // Tree elements
        public Modifiers modifiers;
        public final Id id;
        public Optional<Id> parent;
        public final List<Field> fields;
        // For convenience
        public final String name;

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

```
public static class MethodDef extends Field {
        // Tree elements
        public Modifiers modifiers;
        public Id id;
        public TypeLit returnType;
        public List<LocalVarDef> params;
        public Block body;
        // For convenience
        public String name;
```

```
        public MethodDef(boolean isAbstract, boolean isStatic, Id id, TypeLit
returnType, List<LocalVarDef> params, Block body, Pos pos) {
            super(Kind.METHOD_DEF, "MethodDef", pos);
            this.id = id;
            this.returnType = returnType;
            this.params = params;
            this.body = body;
            this.name = id.name;
            if (isStatic)
                this.modifiers = new Modifiers(Modifiers.STATIC, pos);
            else if (isAbstract)
                this.modifiers = new Modifiers(Modifiers.ABSTRACT, pos);
            else
                this.modifiers = new Modifiers();
        }
```

```
public static class Modifiers {
        public final int code;
        public final Pos pos;
        private List<String> flags;
        // Available modifiers:
        public static final int STATIC = 1;
        public static final int ABSTRACT = 2;

        public Modifiers(int code, Pos pos) {
            this.code = code;
            this.pos = pos;
            flags = new ArrayList<>();
            if (code==1) flags.add("STATIC");
            if (code==2) flags.add("ABSTRACT");
        }
```

2.局部类型推断：加入 var 关键字，用来修饰局部变量。

为 `Decaf.jacc` 中的 `SimpleStmt` 添加相应的文法。

```
 |   VAR Id '=' Expr
                    {
                        $$ = svStmt(new LocalVarDef(null, $2.id, $3.pos,
Optional.ofNullable($4.expr), $2.pos));
                    }
```

修改 `tree.java` 中相应的 `LocalVarDef` 类。当变量类型为 `var` 时返回 `Optional.empty()`。

```
@Override
        public Object treeElementAt(int index) {
            return switch (index) {
                case 0 -> var==true? Optional.empty():typeLit;
                case 1 -> id;
                case 2 -> initVal;
                default -> throw new IndexOutOfBoundsException(index);
            };
        }
```

3.First-class Functions

为了实现这个新特性，需要实现以下三个方面：

（1）添加函数类型

在`Decaf.jacc` `Type`中添加新的文法，并且需要新添加`TypeList`。

```
|   Type '(' TypeList ')'
{
$$ = svType(new TLambda($1.type, $3.typeList, $1.pos));
}
```

```
TypeList         :   TypeList1
                     {
                         $$ = $1;
                     }
                 |   /* empty */
                     {
                         $$ = svTypes();
                     }
                 ;

TypeList1        :   TypeList1 ',' Type
                     {
                         $$ = $1;
                         $$.typeList.add($3.type);
                     }
                 |   Type
                     {
                         $$ = svTypes($1.type);
                     }
                 ;
```

相应的在`tree.java`中添加`TLambda`类进行解析

```
    public static class TLambda extends TypeLit {
        // Tree element
        public TypeLit returnType;
        public List<TypeLit> params;

        public TLambda(TypeLit typeLit, List<TypeLit> params, Pos pos) {
            super(Kind.T_LAMBDA, "TLambda", pos);
            this.returnType = typeLit;
            this.params = params;
        }

        @Override
        public Object treeElementAt(int index) {
            return switch (index) {
                case 0 -> returnType;
                case 1 -> params;
                default -> throw new IndexOutOfBoundsException(index);
            };
        }
```

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

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

（2）lambda表达式

在`Decaf.jacc` `Expr`中添加新的文法。

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

在`tree.java`中新增`LambdaBlock`和`LambdaExpr`类，构造函数分别为

```
        public LambdaBlock(List<LocalVarDef> params, Block body, Pos pos) {
            super(Kind.LAMBDA, "Lambda", pos);
            this.params = params;
            this.body = body;
        }
```

```
        public LambdaExpr(List<LocalVarDef> params, Expr expr, Pos pos) {
            super(Kind.LAMBDA, "Lambda", pos);
            this.params = params;
            this.expr = expr;
        }
```

（3）函数调用

修改`Decaf.jacc` `Expr`中的Call语句，receiver id为Expr。

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

相应的修改`tree.java`中的`Call`类，其构造函数为

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

三、实验思考

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：语法上表现为存在产生式B->A, 即A是B在语法分析树中的儿子节点。

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：框架与if/else相关的产生式可以概括为：S -> ε| iSE ， E -> eS| ε ，如果遇到if ...if...else ...的情况，else会以就近原则匹配最接近的if，从而消除了二义性问题。

Q3. PA1-A 在概念上，如下图所示：

输入的程序 --> lexer --> 单词流 (token stream) --> parser --> 具体语法树 (CST) --> 一通操作 --> 抽象语法树 (AST)

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：本框架中CST和AST是同时构造的，yysv数组为CST，而构造yysv数组时需要产生AST上节点，两者最终同时产 生。

四、实验小结

这是我接触编译的第一次试验，感觉过程比较艰难，主要是直接面对一份框架，和各种嵌套的链接，实在无从下手，我也在反思我自己的理解能力，但还是希望能有一份完整明确的说明，哪怕长一点也还好。在摸索着开始之后，其实主要是模仿着在写，所以有一些bug的调试很不顺利，完全是靠一点一点悟。。。到PA1-A全部写完，再复盘分析，我才对这次实验有了一个整体的认识，回过头来看实验指导，也觉得写的还算明白，所以感觉刚开始的时候主要是看不到重点。希望下一次实验能顺利一些吧。