PA1-A实验报告

计73 林俊峰

2017011303

实验目的

编码实现 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。

0 通用操作

0.1 新增关键字(以"abstract"为例)

1. `Tokens.java` 中注册关键字

   ```
   int ABSTRACT = 200;
   ```

2. `Decaf.jfex` 中注册关键字

   ```
   "abstract" { return keyword(Tokens.ABSTRACT); }
   ```

3. `Decaf.jacc` 中引入新的token

   ```
   %token ABSTRACT
   ```

4. `JaccParser.java` 中注册关键字

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

5. `SemValue.java` 中注册关键字

   ```
   case Tokens.ABSTRACT -> "keyword : abstract";
   ```

0.2 新增操作符

除了新增关键字的操作以外，在 `Decaf.jacc` 的声明段声明操作符的优先级和结合顺序

0.3 修改文法

在 `Decaf.jacc` 中修改文法，有可能要修改语法树 `tree.java`

1 抽象类/方法

问题描述

加入 `abstract` 关键字，用来修饰类和成员函数

语法规范：将原来的

```
classDef ::= 'class' id  ('extends' id)?  '{' field* '}'
methodDef ::= 'static'? type id '(' paramList ')' block
```

变成

```
classDef ::= 'abstract'? 'class' id  ('extends' id)?  '{' field* '}'
methodDef ::= 'static'? type id '(' paramList ')' block
            | 'abstract' type id '(' paramList ')' ';'
```

实现思路

1. 新增 `abstract` 关键字,
2. 修改对应文法
3. 在 `Tree.java` 中修改 `Modifiers` 类

具体实现就是模仿 `static` 关键字来写，比较简单

文法实现

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

分析树实现

修改 `ClassDef` 类的构造函数为

```
public ClassDef(boolean isAbstract, Id id, Optional<Id> parent,
List<Field> fields, Pos pos){...}
```

修改 `MethodDef` 类的构造函数为

```
public MethodDef(boolean isAbstract, boolean isStatic, Id id,
TypeLit returnType, List<LocalVarDef> params, Optional<Block>
body, Pos pos) {...}
```

通过增加的参数来指示类/方法是否带有修饰符

2 局部类型推断

问题描述

加入 `var` 关键字，用来修饰**局部变量**

语法规范：:

```
simpleStmt ::= ...
               | 'var' id '=' expr
```

实现思路

1. 新增 `var` 关键字

2. 修改对应文法，模仿 `Var Initializer` 的文法，但是要修改对应类型为空 `Optional.empty()`

文法实现

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

3 First-class Functions

3.1 函数类型

问题描述

新增语法规范：

```
type ::= ...
            | type '(' typeList ')'
typeList ::= (type (',' type)*)?
```

实现思路

无需注册新的关键字，但是要新建类型

1. `Tree.java` 新增 `TLambda` 类型，继承基类 `TypeLit`，包含两个成员变量(返回类型和参数列表)
2. `Decaf.jacc` 修改文法

- 由于这里主要一个形如 `(int, int, int)` 的类型列表，我模仿参数列表类 `svVars` 在 `AbstractParser` 中新建了 `svTypes`，模仿参数列表的文法VarList 在 `Decaf.jacc` 中新增了 `TypeList`

文法实现

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

TypeList        :   TypeList1
                    {
```

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

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

AbstractParser.java

```
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;
}
```

语法树实现

```
public static class TLambda extends TypeLit {
   public TypeLit returnType;
   public List<TypeLit> param;
   public TLambda(TypeLit returnType,List<TypeLit> param, Pos
pos) {
       super(Kind.T_LAMBDA, "TLambda", pos);
       this.returnType = returnType;
       this.param = param;
   }

   @Override
   public Object treeElementAt(int index) {
```

```
        return switch (index){
            case 0 -> returnType;
            case 1 -> param;
            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);
    }
}
```

3.2 Lambda 表达式

问题描述

新增语法规范

```
expr ::= ...
            | 'fun' '(' paramList ')' '=>' expr
            | 'fun' '(' paramList ')' block
paramList ::= (type id (',' type id)*)?
```

新增关键字 'fun'

新增操作符 '=>', 优先级最低

实现思路

- Decaf.jacc 修改对应文法
- 新增操作符 '=>' 设最低优先级
- 新增关键字 'fun'
- 注意区分Expr Lambda和Block Lambda

文法实现

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

分析树实现

```
public static class Lambda extends Expr {

   public List<LocalVarDef> varList;
   public Expr expr;
   public Block block;

   public Lambda(List<LocalVarDef> varList, Expr expr, Pos pos){
       super(Kind.LAMBDA, "Lambda", pos);
       this.varList = varList;
       this.expr = expr;
       this.block = null;
   }

   public Lambda(List<LocalVarDef> varList, Block block, Pos pos)
{
       super(Kind.LAMBDA, "Lambda", pos);
       this.varList = varList;
       this.block = block;
       this.expr = null;
   }

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

   @Override
   public Object treeElementAt(int index) {
       return switch(index){
           case 0 -> varList;
           case 1 -> block==null? expr:block;
           default -> throw new IndexOutOfBoundsException(index);
       };
   }
```

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

3.3 函数调用

问题描述

语法规范：将原来的

```
call ::= (expr '.')? id '(' exprList ')'
```

变为

```
call ::= expr '(' exprList ')'
```

实现思路

- `Decaf.jacc` 修改语法
- 修改对应的构造函数

实现较为简单，见代码，此处略

问题解答

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？

答: 产生了形如B->A的产生式，在语法树中A是B的子节点

Q2: 原有框架是如何解决空悬 else (dangling-else) 问题的？

答: 通过语法设定优先级解决: `else` 与最近的 `if` 匹配。

Q3: PA1-A 在概念上，如下图所示：

```
作为输入的程序（字符串）    --> 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？

答: 实现过程没有具体语法树, jacc直接构建抽象语法树, 具体语法树仅作为表示源文本的解析结构的概念实体存在