Decaf PA1-A 实验报告

计76　　张翔　　2017011568

2019 年 10 月 7 日

1　实验简述

本阶段的实验要求利用jflex和jacc实现Decaf语言的词法和语法分析，同时生成AST。现有的框架已经支持语言的基本特性，本阶段需要支持三个新增特性：抽象类、局部类型推断以及一等函数（First-class Function）。

2　主要工作

需要改动的部分为：向词法文件中添加关键字、向语法文件中添加新的语法规则、为AST添加新特性对应的节点类型或者修改已有的节点。

2.1　抽象类

此处需要添加关键字`abstract`，直接在`Decaf.jflex`中添加一条规则即可，`Decaf.jacc`中需要添加相应的Token。抽象语法树的实现中已有一`Modifier`类用于处理`static`修饰符，在此先扩展其功能以支持`abstract`修饰符。

注意到对于类定义，只需要将文法改成

```
classDef ::= 'abstract'? 'class' id  ('extends' id)?  '{' field* '}'
```

实际转换为`jacc`格式时，仿造对`extends`的处理方法，增加一个`AbstractClause`，并根据是否匹配到`abstract`关键字分别处理。对于AST的`ClassDef`节点，使用修改后的 `Modifier`来记录是否为抽象类。

对于方法定义，在文法中增加产生式

```
methodDef ::= 'abstract' type id '(' paramList ')' ';'
```

注意到这里并不需要对`abstract`方法却有实现的语法错误进行特殊处理，因为该产生式已经保证`abstract`方法不会有`block`块。在语法树节点`MethodDef`中，则只需要将方法体改成`Optional<Block> body`即可。

2.2　局部类型推断

类似上面的操作，先增加关键字`var`，然后增加产生式

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

注意这里在写入jacc文件时，不能将变量初始化部分替换为Initializer。规则文件中已有的

```
SimpleStmt      :   Var Initializer
```

仅适用于非自动推导的类型，而Initializer可以产生ε，Var既可以是局部变量声明（可以不初始化），也可以是函数参数表中的变量声明。而自动类型推导则必须依靠变量初始化时提供的右值类型，因此不允许有空的初始化语句，同时它只能作用于局部变量而不能出现在函数的参数列表中。实现时直接新增VAR Id '=' Expr即可满足这些要求。

在做类型推导前，var的类型字面量应为空，从而只需要将LocalVarDef节点构造函数中TypeLit改成Optional<TypeLit>即可。

2.3 一等函数

这里的一个核心是对Lambda表达式的支持，需要增加关键字fun以及箭头操作符=>。从语义上来说，Decaf的Lambda表达式和JavaScript几乎是一致的，即=>是右结合的，因此

```
var lambda = fun (int x) => fun(int y) => x-y;
```

应等效于

```
var lambda = fun (int x) => (fun(int y) => x-y);
```

这里可以通过jacc的%right来指示运算符的结合性，而由于箭头操作符优先级最低，需要将其放在优先级表的最前面（优先级从低到高）。

实现时为Lambda表达式添加了一个新的语法树节点类型，继承Expr类型，构造函数接受参数表和返回体，而由于返回体可以是Expr或者Block，因此使用Optional装载它们。

Lambda表达式或者其他普通的函数都可以归为**函数类型**，其包含了返回值类型和参数类型列表的信息，参数类型列表TypeList在jacc中的写法可以完全参照VarList的写法（实质上就是移除了VarList中的变量名），在语法树中使用List<TypeLit>装载即可，而函数类型本身作为一种类型，则需要为语法树新增一个节点类型，实现时新增了继承TypeLit的TLambda类。这里有一个小的建议，样例中输出中**函数类型**在语法树中的名字为TLambda，但本身这种类型也是适用于类成员函数的，改成TFunc等可能会更普适一些。

函数调用的语法根据实验指导书上的信息更改即可。看起来这里将语法做了较大扩展，Expr类均可以作为被调用的候选者，而无需关注这是Lambda表达式、成员函数抑或是其他什么，后面做Type Check时只要检查到这个Expr是函数类型并匹配参数类型后就可以正常调用了。

3 遇到的问题

1. jacc不能直接返回boolean或者List类型，而需要使用SemValue来包装，因此后来新增了svBoolean（用于前面提到的AbstractClause的返回值）和svTypes（函数类型中的参数类型列表）

2. 对于箭头操作符的结合性，测试使用%left或者%nonassoc也都可以正常工作，原因应该是这种语法只有按右结合才能够成功推导（而不像产生式$E \to E + E$ 会存在结合顺序的歧义）。但为了形式上的正确性，jacc规则中还是应该写作%right。

4 正确性测试

所给样例测试全部通过，另外也测试了下面的一些情况：

1. 函数类型的多层嵌套：

```
void(int(int,int,int,int),void(string,string,int)) func;
int(int,void(int,string(int))) f;
```

2. 使用`var`时不做变量初始化、`var`出现在参数列表中、`static`与`abstract`混合使用

   这些情况均会报语法错误

5 思考题

- **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？**

  一般来说，语法上会存在形如$B \to \alpha A \beta$的产生式，或者有多步推导关系$B \overset{*}{\Rightarrow} \alpha A \beta$ 。如`Expr`子类有`Lambda`, 各种`TypeLit`, `IntLit`等，而对应也有产生式`Expr` $\to$ `Literal`等。α, β为AST不感兴趣的符号，如括号等。

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

  通过指定优先级的方式来避免dangling-else造成的shift-reduce冲突，注意到优先级表中`ELSE`优先级高于`EMPTY`，因此如果`ElseClause`遇到else时会使用`ELSE Stmt`，从而将其与最近的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） $\xrightarrow{\texttt{lexer}}$ 单词流 (token stream) $\xrightarrow{\texttt{parser}}$ 具体语法树(CST) $\to$ 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？

  框架实现时并不是按照概念模型中的顺序处理操作，而是由parser按需调用lexer获取token，然后直接生成AST。具体语法树在此成为了解析过程的一个概念实体，即parser的工作流程实际上对应了CST的遍历，它在CST的某些节点（如此前遍历的节点刚好对应某个产生式）上进行操作，生成相应AST节点。