Decaf PA1-B 实验报告

2017011462 方言

一、新特性：

这部分的`Decaf.jflex`文件沿用PA1-A，`Tokens`和`Parser`中注册新关键字的方法与PA1-A中相同，不再赘述。

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

- `ClassDef`的修改方法与PA1-A几乎一致，在此不赘述。
- `FieldList`中，增加新的产生式，同时要保证符合LL(1)文法的要求。语义动作里将新的节点加在后续List的头部（这里参考其他产生式的语义动作实现）

```
ABSTRACT Type Id '(' VarList ')' ';' FieldList
{
$$ = $8;
$$.fieldList.add(0, new MethodDef(false, true, $3.id, $2.type, $5.varList,
Optional.empty(), $3.pos));
}
```

由此即可完成特性一，难度并不是很大。

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

- 在`Decaf.spec`中，在`SimpleStmt`中加入新的产生式和语义动作：

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

- 在`Tree.java`的`LocalVarDef`类中：
  - 与PA1-A一样，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)
```

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.spec`中：
  - 在`Type`中，增加新的语义动作

    ```
    AtomType ArrayType
    {
        $$ = $1;
        for (var sv : $2.thunkList)
        {
            if(sv.typeList == null)
                $$.type = new TArray($$.type, $1.type.pos);
            else
                $$.type = new TLambda($$.type, sv.typeList, $1.type.pos);
        }
    };
    ```

    这里根据是否有`SemValue`是否有`typeList`，来判断其是不是`TLambda`类型，从而采用不同的构造函数。
  - 在`ArrayType`中，增加新的产生式。

    ```
    '(' TypeList ')' ArrayType
    {
    var sv = new SemValue();
    sv.typeList = $2.typeList;

    $$ = $4;
    $$.thunkList.add(0, sv);
    }
    ```

    `ArrayType`实际上是为了消除直接左递归而产生的，因此写产生式时也要注意这一点。另外将新的`SemValue`加入后续`thunkList`的头部（实际上就是从左到右加入）
  - 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：
  - 与PA1-A一样，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`
  - 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`

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：

- 增加新的产生式（注意要提取左公因子）：

```
Expr            :   FUN '(' VarList ')' AfterRParen
                    {....}

AfterRParen     :   ARROW Expr
                    {
                        $$ = new SemValue();
                        $$.expr = $2.expr;
                    }
                |   Block
                    {
                        $$ = new SemValue();
                        $$.block = $1.block;
                    }
                ;
```

  语义动作中，通过判断`expr`是否为空，来区分两种不同的结构。

- 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`

3.3 函数调用

- 在`ExprT8`中增加新的产生式，参考其他产生式实现。

```
'(' ExprList ')' ExprT8
{
    var sv = new SemValue();
    sv.pos = $1.pos;
    sv.exprList = $2.exprList;
    $$ = $4;
    $$.thunkList.add(0, sv);
}
```

- 在`Expr8`中修改语义动作

```
{
    $$ = $1;
    for (var sv : $2.thunkList) {
        if (sv.expr != null)
        {
```

```
            $$ = svExpr(new IndexSel($$.expr, sv.expr, sv.pos));
        }
        else if (sv.exprList != null)
        {
            if (sv.id == null)
                $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
            else
            {
                $$ = svExpr(new VarSel($$.expr, sv.id, sv.id.pos));
                $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
            }
        }
        else
        {
            $$ = svExpr(new VarSel($$.expr, sv.id, sv.pos));
        }
    }
    $$.pos = $$.expr.pos;
}
```

原先我并没有增加新的分支判断`id`，在与标准输出对比后发现，比标准输出少了一行`VarSel`，并且其内容是`method id`。因此增加了一个新分支，判断`SemValue`中是否有`id`，如果有的话，就先增加一层`expr`，将`id`指定为`VarSel`，再把这个`expr`传给`Call`，这样可以达到标准输出的结果。

- 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。

到这里已经与标准输出一致，但是`gradle build`时发现`ExprListOpt`处有冲突，询问同学后，我发现并没有其他地方使用到`ExprListOpt`，于是我直接去掉了`ExprListOpt`（`Expr9`中有使用，也将那一部分去除，并对语义动作做相应修改）这样解决了冲突。

由此即可完成特性三的实现。

二、错误恢复

在`parseSymbol`中增加如指导书中所述的流程。

- 构造`Begin`和`End`集，直接调用`LLTable.java`中的`beginSet`和`followSet`方法，再将原先的`follow`集加入`followSet()`中构成`End`集。
- 如果`token`（实际上就是`lookahead`）不在`Begin`集中，则认为出错，报错，进入分析流程。
- 若`token`为`Begin`集中的符号，则可恢复分析，得到`result`。若为`End`集中的符号，则认为该次分析失败，返回`null`。其他符号均跳过（即不做处理）
- 最后，当没有出错时调用`act`，可通过判断`issuer`中的`errors`是否为空。（报错使用`yyerror`，可以看到其生成了`DecafError`实例，并将其加入`errors`中）

  需要注意的是，在递归分析时，传入的`follow`集应为生成的`End`集。

三. 问题

Q1：本阶段框架是如何解决空悬 else (dangling-else) 问题的?

答：通过查看`LLTable.java`中可以看到，在处理`ElseClause`的分支时，出现`ELSE`时即进入`ELSE`分支，而其他情况（对应`ELSE`为空），进入另一分支。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。

Q2：使用 LL(1) 文法如何描述二元运算符的优先级与结合性？请结合框架中的文法，举例说明。

答：优先级通过不同产生式来描述。比如`Expr2`是处理`Op1`这个优先级的，产生式`ExprT1 --> Op1 Expr2 ExprT1`，`Expr2 --> Expr3 ExprT2`和`ExprT2 --> Op2 Expr3 ExprT2`可以看出，外部先分析`Op1`，而内部再分析`Op2`，因此`Op1`的优先级低于`Op2`。以此类推，`Op1`到`Op7`优先级从低到高。

在LL(1)文法中结合性应当是右结合的，框架中使用一个list从左到右加入节点，之后再分析，可认为是左结合的。比如在产生式`ExprT1 --> Op1 Expr2 ExprT1`的语义动作中，将`Expr2`生成的节点加入到右侧`ExprT1`的`thunkList`队头，实际上就是从左到右记录。

Q3：无论何种错误恢复方法，都无法完全避免误报的问题。 请举出一个具体的 Decaf 程序（显然它要有语法错误），用你实现的错误恢复算法进行语法分析时会带来误报。 并说明该算法为什么无法避免这种误报。

```
class Main {
    abstract bol test();
    static void main() {
    }
}
```

在`bool test()`中去掉一个`o`，应当只在此处报错，但实际报错如下

```
*** Error at (2,14): syntax error
*** Error at (2,18): syntax error
```

由于没有关键字`bol`，根据错误恢复的流程，会在处理`Type`时报错，并且会将其视为`id`算在`Type`的`End`集之中。因此会认为`Type`分析失败，`bol`被识别为`id`，由此`test`会被认为对应于`(`，因此会再次报错。