Decaf PA1-B 实验报告

计 76　张翔　2017011568

2019 年 10 月 25 日

1　实验简述

本阶段的实验要求与上阶段类似，但 Parser 框架换成 `ll1pg`，需要在 `decaf.spec` 中加入新特性的 $LL(1)$ 文法，并在 `LLParser.java` 中实现错误恢复。由于我选择在整体框架上开发，PA1-A 的代码可以直接复用，增加了便利性。

2　主要工作

2.1　$LL(1)$ 文法变换

2.1.1　抽象类 本地类型推导

对于抽象类和本地类型推导，由于引入了新的关键字 `abstract` 和 `var`，可以很容易地在现有的产生式中加入新项而保持 $LL(1)$ 性质。`ClassDef` 文法与上阶段相同，可直接添加一个 `AbstractClause`；PA1-A 的 `FieldList` 存在左递归 `FieldList`→ `FieldList MethodDef`，变换后出现 `FieldList` → `MethodDef FieldList`，当然这里非 `abstract` 或 `static` 的方法与变量声明 `Type Id` 之间存在左公因子，也同时做了消去，与 `abstract` 互不干扰，因此直接在新文法的 `MethodDef` 添加抽象方法的产生式即可。对于 `var` 的操作也是同理。

2.1.2　Lambda 表达式

相比前面两个新特性，实现 Lambda 表达式相关的内容会更加复杂。首先需要添加

```
Expr ::= 'fun' '(' Varlist ')' ('=>' Expr | Block ) | Expr '(' ExprList ')'
```

其中 Lambda 表达式优先级最低，而函数调用优先级仅次于最高的括号（指`'(' Expr ')'`）。变换 $LL(1)$ 文法时可将原来的文法写成优先级文法，然后消左递归与左公因子。优先级变换可以采用算符优先级联的形式改写文法，如 Decaf 有 10 种算符优先级，除去优先级最低的 Lambda 表达式，前 6 个优先级都是二元运算符，为了方便，记作 `Op_i` ($i = 1, 2, \cdots, 6$)。对于二元运算符，可以改写文法

```
Expr ::= Expr Op_i Expr
```

为

```
Expr_i ::= Expr_i Op_i Expr_i | Expr_(i+1)
```

然后可以根据结合性来消除二义性，如左结合

$$\texttt{Expr\_i} ::= \texttt{Expr\_i Op\_i Expr\_(i+1)} \mid \texttt{Expr\_(i+1)}$$

不结合

$$\texttt{Expr\_i} ::= \texttt{Expr\_(i+1) Op\_i Expr\_(i+1)} \mid \texttt{Expr\_(i+1)}$$

采用这种方法，可以改写 Decaf 的 `Expr` 文法为

$$\begin{aligned}
\texttt{Expr} &\to \texttt{E1} \mid \texttt{LambdaExpr} \\
\texttt{E1} &\to \texttt{E1 OP1 E2} \mid \texttt{E2} \\
\texttt{E2} &\to \texttt{E2 OP2 E3} \mid \texttt{E3} \\
&\vdots \\
\texttt{E4} &\to \texttt{E5 OP4 E5} \mid \texttt{E5 // OP4 is non-associative} \\
&\vdots \\
\texttt{E7} &\to \texttt{OP7 E7} \mid \texttt{(Class Id) E7} \mid \texttt{E8} \\
\texttt{E8} &\to \texttt{E8 (ExprList)} \mid \texttt{E8 [Expr]} \mid \texttt{E8 . Id} \mid \texttt{E9} \\
\texttt{E9} &\to \texttt{(Expr)} \mid \texttt{Final}
\end{aligned}$$

`Final` 为 `this`, `Literal`, `new` $\cdots$ 等相对简单的表达式。注意到 `OP4` 为文法规范中规定的不等比较符号，是不结合的，即 `a < b < c` 这种表达式不符合语法。但 PA1-A 使用的 `jacc` 存在 bug，虽然用`%noassoc` 定义了这一等级运算符的结合性，但实际上 `jacc` 以右结合的方式处理，并没有对这种情况报语法错误。PA1-B 框架 `ll1pg` 文法部分也没有正确处理，而是把它当成了左结合。我实现时做了微小的修改，这个问题得以解决。

之后可以通过消左递归、左公因子得到 $LL(1)$ 文法，注意有间接左公因子的存在，如消除左递归后得到

$$\begin{aligned}
\texttt{Expr} &\to \texttt{E1} \mid \texttt{LambdaExpr} \\
\texttt{E1} &\to \texttt{E2 ET1} \qquad \texttt{ET1} \to \texttt{OP1 E2 ET1} \\
\texttt{E6} &\to \texttt{E7 ET6} \qquad \texttt{ET6} \to \texttt{OP6 E7 ET6} \mid \varepsilon \\
\texttt{E7} &\to \texttt{OP7 E7} \mid \texttt{(Class Id) E7} \mid \texttt{E8} \\
\texttt{E8} &\to \texttt{E9 ET8} \qquad \texttt{ET8} \to \texttt{(ExprList) ET8} \mid \texttt{[Expr] ET8} \mid \texttt{. Id ET8} \mid \varepsilon \\
\texttt{E9} &\to \texttt{(Expr)} \mid \texttt{Final}
\end{aligned}$$

注意到 $\texttt{E7} \overset{*}{\Rightarrow} \texttt{E9} \Rightarrow$ `( Expr )`，与产生式 `E7 → ( Class Id ) E7` 有左公因子。这里将 `E9` 的 `( Expr )` 提前到 `E7`，即变换为

$$\begin{aligned}
\texttt{E7} &\to \texttt{OP7 E7} \mid \texttt{(Class Id) E7} \mid \texttt{(Expr) ET8} \mid \texttt{E8} \\
\texttt{E9} &\to \texttt{Final}
\end{aligned}$$

之后就可以很方便地消除 `E7` 产生式右部的公因子。

2.1.3 函数类型

这里需要将类型对应的文法改成

$$\texttt{Type} ::= \texttt{Type '(' TypeList ')'} \mid \texttt{Type '[' ']'} \mid \texttt{AtomType}$$

消除左递归很容易，可以得到

$$\texttt{Type} \rightarrow \texttt{AtomType T} \quad \texttt{T} \rightarrow \texttt{[ ] T} \mid \texttt{(TypeList) T} \mid \varepsilon$$

TypeList 的类似于同已有的 VarList。构造 AST 时，需要知道函数类型的返回值、参数列表。从 T → (TypeList) T 的语义动作中可直接得到参数列表，但无法确定返回值类型。由于是自顶向下的语法分析，实现时可以在递归返回时将函数类型压栈（代码中的 thunkList），待到 Type → AtomType T 的语义动作执行时再去修正函数类型的返回值类型。这里需要注意 E9 → New Type [ ] 由于 [的存在会产生冲突，需要进一步变换。

3 思考题

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

  通过优先匹配 else 的方式来解决。ll1pg 对于冲突的产生式会优先选择前面的进行匹配，因此规则文件中将带 else 的产生式放在前面即可。

- **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。**

  详见2.1.2中提到的文法变换部分。

- **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。**

```
1  class Main {
2      String main() {
3          var q = new int[2];
4      }
5  }
```

报错为（后 3 次为误报）

```
*** Error at (2,5): syntax error
*** Error at (3,18): syntax error
*** Error at (4,5): syntax error
*** Error at (5,1): syntax error
```

理论上只应该报错 (2, 5) 位置处的 String，实际上却报了其他错误。根据实验指导中的错误恢复算法，遇到 String 后会忽略之后的 token，一直到 new 之后的 int 处恢复，解析到 2 时认为数组类型中不会出现 IntLiteral，从而 Type 解析失败报错，而之后遇到} 属于 End 集合，因此认为 FieldList 解析失败报错，第 4 行} 与第 2 行的匹配，从而认为第 5 行} 失配，从而报错。因此按此算法，必然报错 4 次且无法避免，其根本原因是跳过部分 token 之后对语言的解析情况发生了预期外的改变。