# Decaf PA 1-B 实验报告*

邓博文

2018 年 11 月 3 日

## 1 本阶段工作

### 1.1 增加错误恢复功能

这一步骤的目的是当输入的 Decaf 程序出现语法错误时，parse 函数还能够对后续的程序继续分析。自己使用的错误恢复方法是说明文档中提供的方法。需要说明的地方有几处。首先是当分析某一非终结符时，如果该终结符对应的产生式右侧有非终结符存在，并且右侧的非终结符分析出错，按照 parse 函数的返回值约定，此时右侧非终结符对应的 params[i] 应该为 null。此后调用 act 函数时，会出现 NullPointerException。因为这种情况需要继续分析，并且左侧非终结符分析也失败了，所以使用 try catch 结构捕获异常并直接返回 null。第二点是错误恢复方法依赖于 follow，begin 集合，当加入新的语法特性时，两个集合都会发生变化。所以在正确地添加完所有语法特性之前，出现某一个 S1-中测例错误是正常的。自己因为没意识到这一点，在阶段二调试了许久，其实 parse 函数修改的没

*使用的 JDK 为 JDK11

有问题，只是因为两个集合中的元素还不对，所以会有误报、漏检现象出现。最后一点是实现 parse 函数时，是通过函数 beginSet(),followSet() 来获得 begin,follow 两个集合的。必须注意的是这两个函数返回的是私有成员 begin,follow 的引用，也就是说 parse 函数中对这两个集合的修改会得到保留。由于 $End(A) = Follow(A) \cup F$，自己一开始没有注意到这个问题，实现时用了

$$Set < Integer > end = followSet(symbol);$$
$$end.addAll(follow);$$

这种写法，导致递归下降分析时随着分析的进行，follow 变量的值发生了改变，进而使得程序无法正确跳过不在 $End(A)$ 中的符号。

## 1.2 增加新特性的 LL(1) 文法

像 scopy,sealed 这样的特性和 PA1-A 中没有区别，只需要添加 token 并在 Tree 中定义或者修改相应节点即可。但像串行条件卫士语句这样含有递归定义的文法，则和 PA1-A 中有明显的不同。YACC 建议尽量使用左递归来定义文法，但是 LL(1) 文法则要求不能有左递归，这就要对文法进行改写。比较方便的方法就是把左递归都修改为右递归的形式。这里有两点需要注意。第一点是对于如下形式的产生式：

$$\begin{aligned} XXXList: \quad & \alpha \\ | \quad & /*empty*/ \\ ; & \end{aligned}$$

应满足 $First(\alpha) \cap Follow(XXXList) = \varnothing$，否则会有冲突产生。第二点是和 PA1-A 不同，因为 SemValue 定义时已经为成员变量 slist 分配了内存，

所以 Parser.spec 中无需再为其分配内存。比较困难的是 Expr 部分。这部分需要理解运算符的优先级和结合性是如何体现的。总的来说，OperX 中每一个 X 对应一类优先级的各个操作符，X 越大优先级越高。如下的文法定义结构：

$$
\begin{aligned}
ExprX: \quad & Expr(X+1)\, ExprTX && (1)\\
;& \\
ExprTX: \quad & OperX\, Expr(X+1)\, ExprTX && (2)\\
| \quad & /*empty*/ \\
;&
\end{aligned}
$$

(1) 式具有递推的结构，其表达的意义是匹配 Expr(X+1)，并优先尝试 ExprTX 对应的 OperX，如果匹配失败，再尝试匹配 Oper(X-1)，这正是优先级的体现。另外，可以看到 (2) 中存在递归的结构，这种结构对应的是同一优先级的操作符重复出现的情况，如 $a \times b \times c$。(2) 会返回一个 svec，其中存储着递归结构每一次匹配的操作符种类。结合性体现在 (1) 对应的 action 中。正常情况下因为是从左至右分析，所以 (1) 处的 action 中如果是对 (2) 返回 svec 顺序访问并创建节点，则是左结合性，因为每个 Expr 都是先和左边的操作符进行结合。数组初始化操作符是右结合性，所以访问 (2) 返回的 svec 时需要逆序访问。最后还要指出的是，虽然文法上定义的 default 是

$$Expr ::= \quad Expr\,[\,Expr\,]\,default\,Expr$$

但由于 default 的优先级是最高的，所以 default 表达式最后的一个 Expr 应为 Parser.spec 中的 Expr9。

## 2 ElseClause 冲突问题

pg.jar 工具有非严格模式和严格模式。在非严格模式下，当同一非终结符对应的两个产生的预测集合之交不为空时，可以通过设置优先级，使得某一产生式优先于另外一个产生式被选择，进而解决选择哪个产生式的问题。具体来说，在 Table.java 中，可以看到 $Follow(ElseClause)$ 中包括 $ELSE$，而 $First(ELSE\ Stmt) = ELSE$。所以若当前非终结符为 ElseClause，并且 lookhead 是 $ELSE$ 时，便有两个产生式可以选择。根据 Parser.spec 中的写法，此时 $ElseClause$ 总是会选择前者。一个存在这种冲突的程序段如图1。

```
class Main {
    static void main() {
      if (true)
        if (false) {

        }
        else
            Print("else has higher priority!\n");
    }
}
```

图 1: 冲突程序片段

此程序片段中，else 语句处存在冲突。按照 Parser.spec 的写法，这个 else 应该和距离它最近的 if 配对，验证其对应的语法树结果见图2。

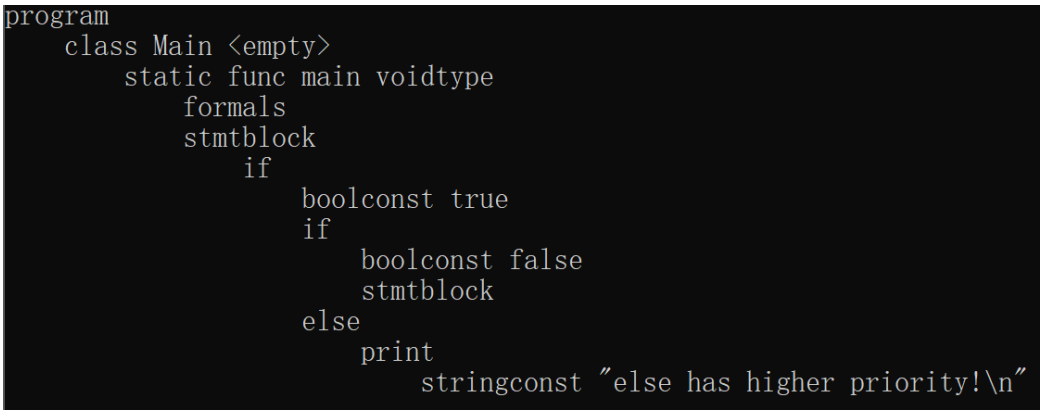


图 2: 冲突程序片段的语法树

一种可能的解决方法如图3。

```
class Main {
    static void main() {
      if (true) {
        if (false) {

        }
      }
        else
            Print("else has higher priority!\n");
    }
}
```

图 3: 可能的解决方法

此时当分析第二个 IfStmt 时，进行到 ElseClause 时，lookhead 将会是}，不会产生冲突，但是第一个 IfStmt 分析到 ElseClause 时仍然存在冲

突。这种写法对应的语法树见图4。

```
program
    class Main <empty>
        static func main voidtype
            formals
            stmtblock
                if
                    boolconst true
                    stmtblock
                        if
                            boolconst false
                            stmtblock
                else
                    print
                        stringconst "else has higher priority!\n"
```

图 4: 解决方法对应的语法树

可以看到，此时 else 是和第一个 if 配对的。

# 3 comprehension 表达式文法

不更改之前 comprehension 表达式和数组常量之间存在冲突。困难的地方在于，即使提取了左公因子 [，因为二者的文法定义中 [之后都可以是 Expr(Constant 也是 Expr)，这就意味着存在着递归的结构。比如说如果连续出现多个 [，那么是无法提前知道需要向前看多少个符号才能确定唯一的产生式的。

# 4 不可避免的误报

一种无法避免的误报情况如图5。

```
class Main {
    static void main() {
        int x} // error 1: this grammar is not supported
        x = 0xf;
        y = 0xff;
        ;
        PrintComp(z,); // error 2: missing parameter after ','
        return 0;
    }
}
```

图 5: 不可避免的误报

此时的报错结果如图6。

```
*** Error at (3,14): syntax error
*** Error at (4,9): syntax error
*** Error at (10,1): syntax error
```

图 6: 误报结果

误报产生的原因如下。因为发现语法错误后跳过符号的个数是由 begin,end 两个集合决定的，所以如果错误符号属于两个集合中的某一个，那么就会无法跳过正确数目的符号。具体来说，这里在 (3,14) 处本来应该是;，但是出现了}，此时正在分析的是非终结符 ArrayType，它会匹配空，进而在返回 VariableDef 时出错，原本此终结符出现错误后应在返回到分析 StmtList 时跳过一定数目的符号，进而继续分析下一个 Stmt，但不幸的是 lookahead 是}，它属于 StmtList 的 follow 集合，所以编译器消耗掉 {，并结束对 StmtList 的分析。随后返回到 FieldList 的分析过程中，lookahead 是 IDENTIFIER( x)，发现 query 函数返回值是 null，报 (4,9) 处错误。这之后

跳过符号直到第 9 行的}，至此结束 FieldList 的分析。当返回到 ClassList 的分析时，发现 lookahead 是第 10 行的}，报 (10,1) 处错误。最后 lookahead 为文件尾，从 ClassList 返回，进而结束整个程序的分析。