

# Decaf PA 1-B 说明

## 任务描述

在 PA1-A 中，我们借助 LEX 和 YACC 完成了 Decaf 的词法、语法分析。在这一部分，我们的任务与 PA1-A 相同，但不再使用 YACC，而是手工实现自顶向下的语法分析，并支持一定程度的错误恢复。

PA1-B 实验的重点是训练自顶向下语法分析/翻译的算法实现。对于词法分析程序，同学们直接利用 PA1-A 的实验结果即可。实验框架默认不包含 Lexer.java 文件，你只需要将 PA1-A 生成的 Lexer.java 文件复制过来即可。

对于语法分析程序，我们提供了 Parser.java 文件作为自顶向下语法分析的模板，其中有部分算法要求大家手工编码实现。而 Parser 类所依赖的 Table 类，由我们提供的 pg.jar 工具自动生成。ant 会自动调用该工具生成 Table 类，你需要通过修改 Parser.spec 文件来得到支持新语法特性的 Table。

本 README 文件只讲述与 PA1-A 有差异的部分，其他信息可参考 PA1-A 的 README 文件。

实验截止时间以网络学堂为准。请按照《Decaf 实验总述》的说明打包和提交。

## 本阶段涉及的工具和类的说明

实验框架与 PA1-A 基本相同，有差别的地方主要是以下几处：

(1) 在 PA1-A 中，我们手工编辑 Lexer.l 文件，再由 JFlex 自动生成 Lexer.java。本次实验框架不包含 Lexer.l 文件，你需要借助 PA1-A 的框架自动生成 Lexer.java，并复制过来。

(2) 在 PA1-A 中，我们手工编辑 Parser.y 文件，再由 BYACC/J 自动生成 Parser.java。本次实验不提供 Parser.y，框架中提供了一份用于手工编码实现 LL(1) 分析的 Parser.java 模板，你需要基于该文件完成本次实验。

(3) 针对 LL(1) 分析所需的数据（包括 First 集合、Follow 集合和预测集合），本框架提供 pg.jar 工具，它根据 Parser.spec 文件所描述的文法，自动生成上述数据，并包装成 Table 类（位于 Table.java 文件）。因此，你只需要在 Parser 类中调用 Table 类中相对应函数，就可以访问到这些数据，从而实现 LL(1) 语法分析。请注意，我们已经在 build.xml 中定义好了调用 pg.jar 工具的逻辑，你只需修改 Parser.spec，然后用 ant 来构建，就能自动生成新的 Table.java。当你的 Parser.spec 描述有误时，pg.jar 工具会报错，请留意构建时工具输出的错误信息。

在 TestCases 目录下，是我们从最终测试集里面抽取出来的一部分测试用例，你需要保证你的输出和我们给出的标准输出是完全一致的。其中 TestCases/S1+ 中的测例保证语法正确，TestCases/S1- 中的测例均有语法错误。

本阶段主要涉及的类和文件如下：

文件/类	含义	说明
BaseLexer	词法分析程序基础	根据 PA1-A 的实现进行修改
Lexer	词法分析器，主体是 yylex()	框架不提供，沿用 PA1-A 自动生成的程序
Parser.spec	LL(1)文法描述	该文件描述了增加新特性前 Decaf 语言的所有文法。你需要先阅读给出的 LL(1) 文法，然后增加新特性对应的 LL(1)文法
Table	语法分析所依赖的数据	由 pg.jar 工具根据文法描述 Parser.spec 自动生成，请勿手工修改
Parser	语法分析器	你需要基于该模板完成本次实验，请先阅读本类定义的所有成员变量和函数，然后实现其中的算法函数 parse
SemValue	文法符号的语义信息	可根据自己的需要进行适当的修改
tree/*	抽象语法树的各种结点	你要在此文件中定义实验新增特性的语法结点，但可以直接将阶段一（A）完成的结果复制过来（注意，是仅复制新增特性的语法结点，而非整个 Tree.java 文件）。
error/*	表示编译错误的类	不要修改
Driver	Decaf 编译器入口	调试时可以修改
Option	编译器选项	不要修改
Location	文法符号的位置	不要修改
utils/*	辅助的工具类	可以增加，但不要修改原来的部分
build.xml	Ant Build File	不要修改

修改好代码后，运行 ant，会在 result 目录下产生 decaf.jar 文件，启动命令行输入 java -jar decaf.jar 就可以启动编译器。不写任何参数的会输出 Usage。

测试和提交方法请参照《Decaf 实验总述》。

## 实验内容与提示

由于本次实验内容较多，且具有一定难度。建议分以下三个步骤来完成本次实验。

## 步骤一：阅读 LL(1)分析算法的实现

在 Lecture04 中，我们介绍了递归下降的 LL(1)分析方法，并给出了一系列非终结符对应的分析函数。例如为了分析文法

$$\langle \text{function} \rangle \rightarrow \text{FUNC ID ( } \langle \text{parameter\_list} \rangle \text{ ) } \langle \text{statement} \rangle$$

我们定义分析函数 ParseFunction:

```
void ParseFunction() {
    MatchToken(FUNC);
    MatchToken(ID);
    MatchToken(LPAREN);
    ParseParameterList();
    MatchToken(RPAREN);
    ParseStatement();
}
```

从 ParseFunction 的实现可以总结出：为了完成对非终结符 $\langle \text{function} \rangle$ 的分析，只需依次对 $\langle \text{function} \rangle$ 产生式右部的各符号进行分析。

- 遇到终结符（如 FUNC）时，调用 MatchToken 函数来匹配；
- 遇到非终结符（如 $\langle \text{parameter\_list} \rangle$ ）时，递归调用它对应的分析函数（如 ParseParameterList）进行分析。

此外，为了将分析出的非终结符 $\langle \text{function} \rangle$ 所对应的分析结果（这里考虑其 AST 结点的值）记录下来，我们可以像 PA1-A 实验那样，用一个 SemValue 类的对象记录每个语法符号对应的结果，然后根据这些结果完成某个用户定义的语义动作。据此，上述分析函数修改为：

```
SemValue ParseFunction() {
    SemValue[] params = new SemValue[6 + 1];
    params[1] = MatchToken(FUNC);
    params[2] = MatchToken(ID);
    params[3] = MatchToken(LPAREN);
    params[4] = ParseParameterList();
    params[5] = MatchToken(RPAREN);
    params[6] = ParseStatement();

    params[0] = new SemValue();
    // do user-defined actions
    return params[0];
}
```

这里，我们用长度比产生式右部符号数多 1 的 SemValue 数组，来缓存对产生式右部各符号进行分析得到的结果（params[1]到params[6]）。然后，我们执行用户定义的语义动作，即用户访问并修改 params 数组中的数据。最终，返回 params[0]作为非终结符 $\langle \text{function} \rangle$ 的分析结果。

以上讨论了非终结符对应于单一产生式的情形。如果某个非终结符对应于多个产生式（即存在多种分析方法），那么我们需要先通过查看 lookahead 符号来决定使用哪一个产生式（哪一种分析方法），然后再利用上述方法分析该产生式右部的符号。在 Java 语言中，我们用一个 switch-case 语句即可实现根据 lookahead 符号选择相应产生式的逻辑。具体例子请见 Lecture04，这里不再赘述。

形如 ParseFunction 的分析函数是类似的，为了避免重复，我们在 Parser 类中把它们统一为通用的 parse 函数，并把非终结符（如<function>）作为第一个参数传入：

```
SemValue parse(int symbol, Set<Integer> follow)
```

其中 symbol 为待分析的非终结符。若分析成功，则返回值存储了 symbol 所对应 AST 结点的值；若分析失败，则返回 null。我们提供的代码框架中实现了不带错误恢复功能的 LL(1) 分析算法，实现思路与

```
SemValue ParseFunction()
```

类似。请注意，框架中给出的实现仅在输入程序语法正确的情况下有效；针对语法错误的程序输入，抛异常是正常现象。在完成步骤二后，你的程序对于即使语法错误的输入也不应该抛异常。请仔细阅读这部分的代码，有必要时插入一些打印语句输出调试信息，以便理解该算法的思路。这一阶段，你无需关心第二个参数 follow。

## 步骤二：增加错误恢复功能

接下来，你需要修改 parse 函数，使其具备错误恢复的功能。即当输入的 Decaf 程序出现语法错误时，它还能对后续的程序继续分析，直至读到文件尾。在 Lecture04 中，我们介绍了应急恢复和短语层恢复的方法。这里，我们提出一种介于二者之间的错误恢复方法：

与应急恢复的方法类似，当分析非终结符  $A$  时，若当前输入符号  $a \notin \text{Begin}(A)$ ，则先报错，然后跳过输入符号串中的一些符号，直至遇到  $\text{Begin}(A) \cup \text{End}(A)$  中的符号：

- 若遇到的是  $\text{Begin}(A)$  中的符号，可恢复分析  $A$ ；
- 若遇到的是  $\text{End}(A)$  中的符号，则  $A$  分析失败，返回 null，继续分析  $A$  后面的符号。

这个处理方法与应急恢复方法的不同之处在于：

- 我们用集合  $\text{Begin}(A) = \{s \mid M[A, s] \text{非空}\}$ （其中， $M$  为预测分析表）来代替  $\text{First}(A)$ 。由于  $\text{First}(A) \subseteq \text{Begin}(A)$ ，我们能少跳过一些符号。
- 我们用集合  $\text{End}(A) = \text{Follow}(A) \cup F$ （其中， $F$  为 parse 函数传入的第二个参数）来代替  $\text{Follow}(A)$ 。由于  $F$  集合包含了  $A$  各父节点的 Follow 集合，我们既能少跳过一些符号，同时由于结束符必然属于文法开始符号的 Follow 集合，本算法无需额外考虑因读到文件尾而陷入死循环的问题。这个处理方法借鉴了短语层恢复中 EndSym 的设计。

另外，当匹配终结符失败时，只报错，但不消耗此匹配失败的终结符，而是将它保留在剩余输入串中。这部分的处理已经在 matchToken 函数中实现。

错误恢复中的一个难题是，某一处的语法错误可能带来后续多处的误报。本实验，我们并不要求你的分析程序多么完美，能达到多低的误报率，而是希望能避免某些典型情形下的误报。只要你的程序针对 TestCases/S1-给出的所有测例，既不漏报错，也不多报错，就视为满足实验要求。为了达到这一目标，你既可以按照上文提出的策略来实现错误恢复，也

可以自己提出一个策略来实现错误恢复。如果是后者，我们允许你根据实际编码的需求**任意修改** Parser 类，请务必在**实验报告**中清晰地描述出你的策略以及对 Parser 所作的修改。

### 步骤三：增加新特性对应的 LL(1) 文法

在正确实现 LL(1) 分析算法之后，我们就能通过在 Parser.spec 中添加新的文法，调用 pg.jar 工具自动完成 Table 类数据的更新，从而实现对新特性的语法分析支持。Parser.spec 文件的格式与 Parser.y 类似，Spec 文件的文法规范参见工具的 wiki (<https://github.com/paulzfm/LL1-Parser-Gen/wiki/1.-Specification-File>)。你需要做的修改主要分为两个部分：一是在 token 段加上新增的终结符；二是加上新特性对应的 LL(1)文法和语义动作。为了确保能正确生成 Table，请不要修改 Parser.spec 中已经给出的部分。

与 PA1-A 不同的是，Parser.spec 要求 LL(1)文法，除了 else 语句处的警告

```
Warning: conflict productions at line ***:
ElseClause -> ELSE Stmt
ElseClause -> <empty>
```

外，在新增文法后，工具不应该再报出任何其他的警告、甚至错误。

注意：针对原 Decaf 语言的文法

```
SimpleStmt ::= LValue = Expr | Call | ε
```

由于将其改写为等价的 LL(1)文法十分复杂，为了简化，本阶段我们讲上述文法扩展为

```
LValue ::= Expr (事实上这一条已经没用了)
```

```
SimpleStmt ::= Expr = Expr | Expr | ε
```

你在进行实验时无需修正此文法。但是之后阶段的作业，请大家以原 Decaf 语言的 LValue 文法为准。AST 打印规范和错误输出格式**同 PA1-A**。

为方便，以下重新列出新增加的语言特性。请注意有两个语言特性的文法与上一阶段有所不同。

#### 1. 支持对象复制语句。

新增对象浅复制语句，形如 **scopy**(id, E)，**scopy** 为新增关键字，语义说明参见后续阶段。

参考语法：

```
Stmt ::= OCStmt ; | ...
OCStmt ::= scopy ( identifier, Expr )
```

#### 2. 引入关键词 **sealed** 修饰 class，使其不能被继承。

参考语法：

```
ClassDef ::= <sealed> class identifier
           <extends identifier> {Field*}
```

#### 3. 支持串行条件卫士语句。 串行条件卫士语句的一般形式如

**if** {  $E_1 : S_1$  |||  $E_2 : S_2$  ||| ... |||  $E_n : S_n$  }

其语义解释为:

- (1) 依次判断布尔表达式  $E_1, E_2, \dots, E_n$  的计算结果。
- (2) 若计算结果为 **true** 的第一个表达式为  $E_k (1 \leq k \leq n)$ , 则执行语句  $S_k$ 。
- (3) 转下一条语句。

参考语法:

```
Stmt ::= GuardedStmt | ...
GuardedStmt ::= if { IfBranch* IfSubStmt } | if { }
IfBranch ::= IfSubStmt |||
IfSubStmt ::= Expr:Stmt
```

4. 支持简单的自动类型推导。

“**var**  $x$ ”可出现在赋值语句的左边, 其类型可由语句有变的表达式类型推导; 或者出现在下面的数组 **foreach** 语句中, 将  $x$  绑定于数组的各个元素, 其类型可由数组元素的类型推导。(新增关键字 **var**)。

赋值语句参考语法:

```
Stmt ::= SimpleStmt ; | ...
SimpleStmt ::= var identifier = Expr | ...
```

注意: 由于我们扩展了 LValue 的定义, 但是我们不希望出现

**Expr ::= var identifier | ...**

因此这里我们限制 **var identifier** 只能出现以赋值语句的左值出现在 SimpleStmt 中。

5. 支持若干与一维数组有关的表达式或语句:

- 1) 数组常量, 形如  $[c_1, c_2, \dots, c_n]$ , 其中  $c_1, c_2, \dots, c_n$  为同一类型的常量。

例如:  $[1, 2, 3]$ ,  $[5]$ ,  $[],$   $[\text{"how"}, \text{"are"}, \text{"you"}]$

参考语法:

```
Constant ::= ArrayConstant | ...
ArrayConstant ::= [ Constant*, ]
```

- 2) 数组初始化常量表达式, 形如

$E \% \% n$

其语义解释为: 返回一个大小为  $n$  的数组常量, 其元素类型同表达式  $E$  的类型, 其每个元素的值被置为  $E$  的当前取值。

参考语法:

```
Expr ::= Expr %% Expr | ...
```

3) 数组拼接表达式，形如

$$E_1 ++ E_2$$

其语义解释为：把两个（类型相同）数组  $E_1$  和  $E_2$  拼接为一个更长的数组。

参考语法：

$\text{Expr} ::= \text{Expr} ++ \text{Expr} \mid \dots$

注意： $++$  为右结合， $%%$  为左结合，两者优先级均低于加法和减法，且  $%%$  优先级高于  $++$

4) 取子数组表达式，形如

$$E [ E_1 : E_2 ]$$

其语义解释为：从数组  $E$  取出下标位于闭区间  $[E_1 : E_2]$  的一段元素构成子数组。

参考语法：

$\text{Expr} ::= \text{Expr} [ \text{Expr} : \text{Expr} ] \mid \dots$

5) 数组下标动态访问表达式，形如：

$$E [ E_1 ] \text{ default } E'$$

其中， $E_1$  为整数类型表达式， $E$  为数组类型的表达式， $E'$  为表达式， $E'$  与  $E$  的元素具有同样类型。

其语义解释为：若  $E_1$  的计算结果为数组  $E$  的合法下标，则返回该数组相应元素的值；否则，则返回表达式  $E'$  的值。

参考语法：

$\text{Expr} ::= \text{Expr} [ \text{Expr} ] \text{ default Expr} \mid \dots$

注意： $\text{Expr} [ \text{Expr} ] \text{ default Expr}$  应看作一个三元操作符，其优先级高于其他一元、二元操作符。

例 1:  $a[3] \text{ default } 2 \% \% 2 ++ [1]$  对应  $(a[3] \text{ default } 2) \% \% 2 ++ [1]$

例 2:  $a[3] \text{ default } 2 + 3$  对应  $(a[3] \text{ default } 2) + 3$

例 3:  $a[3] \text{ default } 2$  对应  $-(a[3] \text{ default } 2)$

6) Python 风格的数组 comprehension 表达式，形如

$$[ [ E' \text{ for } x \text{ in } E \text{ if } B ] ]$$

或者当  $B$  恒为 `true` 时，简写为

$$[ [ E' \text{ for } x \text{ in } E ] ]$$

（新增关键字 **in**）。

注意：PA1-A中我们用方括号[E' **for** *x* **in** E **if** B]，但是该语法若改写为LL(1)文法会有难以解决的冲突（请思考为什么）。为了让大家的生活不那么艰难，PA1-B中我们将方括号改为 [ *...* ]，这样实现起来比较简单。相应地，测例文件也做了调整。在修改文法前，你需要在Lexer中新加入终结符 [ 和 ]。

其语义解释为：对于数组 E 中的每个元素 *e*（用变量 *x* 绑定，*x* 的类型与 E 的元素类型一致），若条件 B 为真，更新其为表达式 E'（与 E 的元素类型相同）的取值，并加入到新数组中。

例 1：每个元素加 1： [ *x* + 1 **for** *x* **in** arr ]

例 2：每个大于 5 的元素乘 2： [ *x* \* 2 **for** *x* **in** arr **if** *x* > 5 ]

参考语法：

Expr ::= [ Expr **for** identifier **in** Expr < **if** BoolExpr > ] | ...

7) 数组迭代语句，形如

**foreach** (var *x* **in** E **while** B) S

或者

**foreach** (Type *x* **in** E **while** B) S （这里，Type 为用户指定类型）

（新增关键字 **foreach**, **in**）。

其语义解释为：对于数组 E 中的每个元素 *e*（用变量 *x* 绑定，*x* 的类型可由 E 的元素类型进行推导或者由用户显式指定的类型，后者应当是 E 的元素类型或者其父类类型）按照下标由小到大进行迭代，仅当条件 B 为真时才进行迭代，每次迭代执行代码块 S（其中允许出现 **break**）；若条件 B 为假，立即退出。

参考语法：

Stmt ::= ForeachStmt | ...

ForeachStmt ::= **foreach** ( BoundVariable **in** Expr < **while** BoolExpr > ) Stmt

BoundVariable ::= **var** identifier | Type identifier

本次实验中对文法所做的修改只是为了改写 LL(1)文法时更加方便，后续实验请以 PA1-A 的框架为基准（由于 PA1-A 中个别文法定义不够准确，请以网络学堂里的“新增特性的类型系统和操作语义定义”为准）。

## 关于 pg.jar 工具

本实验所用的工具基于 LL1-Parser-Gen (<https://github.com/paulzfm/LL1-Parser-Gen>)，但是作了修改（见 course 分支：<https://github.com/paulzfm/LL1-Parser-Gen/tree/course>）。原来的工具可以生成完整的 Parser，但是不具备错误恢复的功能，且在生成的代码基础上添加错误恢复的功能十分困难。因此，我们对此工具作了修改，使其能生成 Table 类，用户基于生成的 Table 类，可以更加自由地实现自己的 Parse 函数，并实现错误恢复的功能。感兴趣的同学可以用原工具生成 Decaf 语言的 Parser 代码，并阅读代码，理解 LL(1)分析算法的过程和实现。



## 关于 JDK 版本

经测试，本阶段实验框架可以在 jdk1.8 与 jdk11 下正常编译并运行，我们强烈建议你在 jdk1.8 或 jdk11 下进行本次实验。测评机器上只会安装 jdk1.8 与 jdk11，如果你使用了其他版本的 jdk 进行实验，请确保生成的 decaf.jar 可以在 jdk1.8 或 jdk11 下正常运行，否则会影响实验部分的成绩。

## 实验评分

1. 编码实现部分评分方式与 PA1-A 相同，看所提交程序的输出是否与标准输出**完全一致**，包括一部分隐藏测例。其中语法错误的测例与 TestCases/S1-十分类似，只要你不**采用作弊**的实现方法，如果你的程序能正确处理 TestCases/S1-中的测例，那么它也能正确处理隐藏测例。请注意，针对语法错误的测例，我们依然要求你的输出与标准输出完全一致。本部分占分 **80%**，其中语法正确部分的测例占分 **40%**，语法错误部分的测例占分 **40%**。

2. 实验报告需要包含以下四个内容：

(1) 简要介绍本阶段工作。如果采用了不同于步骤二中提出的错误处理方法，请详细地介绍你的错误处理算法。

(2) Decaf 语言由于允许 if 语句的 else 分支为空，因此不是严格的 LL(1)语言，但是我们的工具依然可以处理这种冲突。请根据工具所生成的预测分析表中 if 语句相关项的预测集合先做猜测，并对照工具 wiki (<https://github.com/paulzfm/LL1-Parser-Gen/wiki/2.-Strict-Mode>)，理解本工具的处理方法。请在实验报告中说明此方法的原理，并举一个具有这种冲突的 Decaf 语言程序片段，说明它哪里有冲突以及如何解决。

(3) 为什么把原先的数组 comprehension 表达式文法

```
Expr ::= [ Expr for identifier in Expr < if BoolExpr > ] | ...
```

改写为 LL(1)比较困难？提示：你可以先按照原先的文法实现，然后观察 pg.jar 的 Warning 输出。

(4) 无论何种错误处理方法，都无法完全避免误报的问题。请举出一个语法错误的 Decaf 程序例子，用你实现的 Parser 进行语法分析会带来误报。根据你用的错误处理方法，这些误报为什么会产生？

本部分占分 **20%**，以上每项内容各占 **5%**。

本阶段实验报告格式要求：

1. 正文字号 10pt-12pt（四号—五号），行距 1-1.5 倍，A4 纸，单栏。总页数上限为 20 页，下限不作要求，但请注意：篇幅过少可能导致问题作答不够清晰，从而影响得分。

2. 考虑到.txt 和.doc(.docx) 格式在不同操作系统上存在编码问题，或者呈现效果不一致，我们要求所有实验报告都采用.pdf 格式。请将它放在 decaf\_PA1\_B/目录下并命名为 report.pdf，运行 submit.py 自动打包。

- Word 用户：导出为 pdf;
- LaTeX 用户：用 pdflatex/xelatex 生成 pdf;

- **Markdown 用户：**建议用 pandoc 转换，可以用命令行选项设置纸张、字号等；
  - **纯文本：**改用上述方法之一。
3. 若实验报告中出现截图，请确保上面的文本、代码清晰可读。
  4. 实验报告中文或英文均可，请注意写上学号和姓名。
  5. 实验报告不得抄袭。若借用了其他同学的想法，或者参考了文献、网络上的资料，请在相应部分加上脚注或引用。