Decaf PA 1-B 说明

任务描述

在 Decaf 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 Decaf PA1-A 相同，但不再使用 YACC，而是通过手工实现自顶向下的语法分析。

阶段一（B）实验的重点是训练自顶向下语法分析/翻译的手工实现。对于词法分析程序有两种选择：同学们可以直接在阶段一（A）实验结果上修改，也可以完全手工实现。前一种方案可以省去不少工作量，然而后一种方案可以使程序的组织结构更加清晰易读。对于前一种方案，只需要将你在阶段一（A）完成的 Lexer.l 复制过来即可，Ant Build 构建工具会自动根据生成 Lexer.java。对于后一种方案，我们目前的实验框架未中提供任何基础代码。

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。关于本次实验更多信息可参见《第五讲课堂讲稿》（Slide05）和《第五讲课堂教案》（Lecture05）。

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，我们手动编辑 Lexer.l 文件，再由 JFlex 自动生成 Lexer.java。本次实验框架中有提供一个基本的 Lexer.l，可用于原始语言的词法分析自动生成，但你需要将你在阶段一（A）完成的 Lexer.l 复制过来。若是完全手工实现词法分析程序，则不用理会这个文件。

(2) 在 PA1-A 中，我们手动编辑 Parser.y 文件，再由 BYACC/J 自动生成 Parser.java。本次实验不提供 Parser.y，框架中提供了一份手工实现的 Parser.java 参考版本，你需要基于该文件完成本次实验。

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。

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	不需要修改，请事先阅读
Lexer.l	LEX 源程序	你需要将阶段一（A）完成的 Lexer.l 复制过来。
Lexer	词法分析器，主体是 yylex()	由 jflex 生成或手工编写。
BaseParser	语法分析程序基础	不需要修改，请事先阅读
Parser	语法分析器，主体是 yyparse()	你需要基于该文件完成本次实验，手工实现针对新语言特征的语法分析功能。注

		意与 Lexer 的衔接，若使用 jflex 生成 Lexer.java，则应将你在阶段一（A）自动生成的 Parser.java 的所有单词 token 的定义照搬过来，替换框架中 Parser.java 的单词 token 定义。
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节点，但可以直接将阶段一（A）完成的结果复制过来。
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，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 Builder，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

本次实验的内容与第一次完全一致，即在所给的基本框架（其中已经完成了《decaf 语言规范》中描述的语法的自顶向下分析）基础上，针对 decaf 语言增加新的语言特性，完成上述实验目标。为方便，以下重新列出新增加的语言特性：

1. 三元运算符：实现三操作数表达式，形如 A ? B : C。其语义解释与 C 语言的条件表达式一致。其优先级仅高于=，比非赋值运算符的优先级都低。运算符结合性为右结合。
   参考语法：
   Expr ::= BoolExpr ? Expr : Expr

2. 一种特殊的二元对象运算，形如 A << B：其语义解释为：A 和 B 为对象，A << B 计算结果返回另外一个对象，其所属类为 A 和 B 所属类的最小公共父类（基于继承层次关系），该对象的成员变量取值为 A 中相应成员变量的取值。假定继承层次关系为自反传递关系，所以 A << B 的所属类也可能是 A 或 B 的所属类。其优先级仅高于=和上述三元运算符。运算符结合性为左结合。

参考语法：

```
Expr  ::=  Expr << Expr
```

3. switch-case 语句：实现 switch-case 控制结构，与 C 语言相同，形如：

```
switch(表达式) {
    case 常量表达式1: 语句块1;
    case 常量表达式2: 语句块2;
    …
    case 常量表达式n: 语句块n;
    default: 语句块n+1;
}
```

参考语法：

```
Stmt   ::=   SwitchStmt
SwitchStmt     ::=    switch ( Expr ) { CaseStmt* <DefaultStmt> }
CaseStmt       ::=    case Constant : Stmt*
DefaultStmt    ::=    default : Stmt*
```

4. repeat-until 循环结构：实现 repeat 循环结构，形如：

```
repeat{
    语句序列
}until（布尔表达式）;
```

参考语法：

```
Stmt   ::=   RepeatStmt ;
RepeatStmt     ::=    repeat Stmt* until ( BoolExpr)
```

5. 循环内部控制语句 continue：其语义是跳过本次循环的后续语句，使控制转移到下一次循环。

参考语法：

```
Stmt   ::=   ContinueStmt ;
ContinueStmt   ::=    continue
```

实验说明和评分

1. 提交的 Parser.java 必须手动编写，不可以使用 BYACC 自动生成的版本代替。使用 BYACC 自动生成的版本代替的不能得分。

2. 可以直接使用 PA1-A 中自动生成的 Lexer.java，也可以重写。直接使用的工作量会小一些；重写可以让 Lexer 和 Parser 都有更清晰易读的代码。

4. 评分方式与 PA1-A 相同，代码部分主要看输出的结果是否正确，包括一部分未公开的测试样例。涉及错误信息的测试样例的评分要求有所放宽（runAll.py 暂未考虑这两点，评分

时会考虑）：

（1）错误信息的（行，列）号要求行号与参考输出一致,形式如：

```
*** Error at (5,15): syntax error
```

（2）错误恢复的问题较为复杂，可以不考虑，你的编译器只要能正确输出第一条错误信息就可以通过测试。