Decaf PA 1-A 说明

任务描述

在本阶段中，大家的任务是编码实现 Decaf 语言编译器的词法分析和语法分析部分，同时生成抽象语法树。

首先你需要运用词法分析程序自动生成工具 Flex 生成 Decaf 语言的词法分析程序。词法分析程序是前端分析的第一部分，它的功能是从左至右扫描 Decaf 源语言程序，识别出标识符、保留字、整数常量、操作符等各种单词符号，并把识别结果以终结符的形式返回给语法分析程序。这一部分的实验目的是要掌握 LEX 工具的用法，体会正规表达式、有限自动机等理论的应用，并对词法分析程序的工作机制有比较深入的了解。

除了词法分析程序以外大家还需要运用语法分析程序自动生成工具 BYACC 生成 Decaf 语言的语法分析程序。在 PA1-A 中，Decaf 语法分析程序的功能是对词法分析输出的终结符串（注意不是字符串）进行自底向上的分析。这一部分的实验目的是要初步掌握 YACC 工具的用法，体会上下文无关文法、LALR(1)分析等理论的应用，并对语法分析程序的工作机制有比较深入的了解。

注意 PA1-A 的两部分是密切相关的：语法分析程序并不直接对 Decaf 源程序进行处理，而是调用词法分析程序对 Decaf 源程序进行词法分析，然后对词法分析程序返回的终结符序列进行归约，也就是说，词法分析程序输出的结果才是语法分析的输入。

最后，当语法分析程序运行结束，语法正确时，PA1-A 会生成与 Decaf 源程序对应的抽象语法树。

PA1-A 是整个实验的热身阶段，需要自己完成的代码量较少，主要的工作在于建立编程环境、熟悉代码框架、熟悉 Decaf 语言以及掌握 Flex 和 BYACC 的具体用法等。

注意，在 BYACC 把语法规则转换为语法分析程序的过程中，可能会报移进/归约冲突或者是归约/归约冲突。不消除这些冲突得到的程序也有可能正常工作，但我们要求必须消除这些冲突。

本阶段的测试要求是输出结果跟标准输出完全一致。我们保留了一些测试例子没有公开，所以请自己编写一些测试例子，测试自己的编译器是符合 Decaf 语言规范里的相应规定。

本阶段持续时间为 2 周，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

实验环境为 JDK1.6 以上版本，词法分析器生成工具为 JFlex，语法分析器生成工具为 BYACC/J，开发环境建议使用 eclipse。请注意 BYACC 本身对%nonassoc 修饰的符号的处理不太正常，因此如果你需要进行语法错误检查的话，请勿涉及无结合性符号方面的错误检查（BYACC 会直接进行归约而不是报错）。

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	你要在此文件中定义正规式，并给出相应的动作。
Lexer	词法分析器，主体是 yylex()	由 jflex 生成
BaseParser	语法分析程序基础	不需要修改，请事先阅读
Parser.y	YACC 源程序	你要在其中加入 Decaf 的语法规则和归约动作
Parser	语法分析器，主体是 yyparse()	由 byacc 自动生成
SemValue	文法符号的语义信息	你要根据自己的需要进行适当的修改
ParserHelper	编写 YACC 动作的辅助类	在这里编写 yacc 的动作，然后粘贴到 Parser.y 中
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
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》。

另外，Lexer 与 Parser 类都提供了 diagnose 函数用于调试，其中 Lexer 是依次输出读到的终结符，而 Parser 是输出归约使用的产生式。

单词符号说明

下面先简单介绍一下 Decaf 语言的单词符号。Decaf 语言的单词符号主要有以下 5 类：

1、关键字：是预先设定的一组字符串，在 Decaf 中关键字同时也是保留字，因此这些字符串不能用作标识符，也不能被重新定义。

2、标识符：是以字母开头，后跟若干字母、数字和下划线字符的串。例如，“int_var”是合法的标识符。需要注意的是 Decaf 语言区分大小写，例如，if 是关键字，而 IF 则是标识符。

3、常量：包括整数、布尔常数、字符串三种。

4、算符和界符（operators and delimiters）：包括单字符的和双字符的两种，其中单字符的算符和界符在词法分析器中直接返回其 ASCII 码即可。

5．注释：Decaf 中只有单行注释（以“//”开始，至行尾结束，如果在程序结尾，则最后需要换行）。

关于单词符号的进一步说明请参考 Decaf 语言规范中的词法规范一节。

实验内容

本次实验将给出 decaf 基本框架，其中已经完成了《decaf 语言规范》中描述的语法的分析以及错误处理。现在，我们对 decaf 语言增加新的语言特性，你需要完成对新语言特性的词法语法分析，以及相关的语法树构建。

增加的语言特性如下：

1. 整复数类型的支持：
   1）新增关键字 **complex**，用于声明复数类型的变量。即，将下列语法规则
   ```
   Type ::= int | bool | string | …
   ```
   修改为
   ```
   Type ::= int | bool | string | complex |…
   ```
   2）同时，应在词法分析中增加识别复数常量虚部的功能。复数表示形式为 a+bj，其中 a 为实部，bj 为虚部，a、b 均为整数
   新增终结符 imgconst 表示复数常量的虚部：
   Constant:: =intconst | boolconst | imgconst | ...
   3）新增表达式：@e 表示获取复数表达式 e 计算结果的实部（整数），$e 表示获取复数表达式 e 计算结果的虚部（整数），#e 表示将整数表达式 e 的计算结果强制转换为复数。（注：本学期，限定复数表达式仅包含加法（+）和乘法（*）运算。）
   参考语法：
   ```
   Expr ::= @Expr | $Expr | #Expr | …
   ```
   4）新增语句复数打印语句（关键字 PrintComp 开头）表示复数的显示，其参数要求具有复数类型。
   参考语法：
   ```
   Stmt ::= PrintCompStmt ; | …
   PrintCompStmt ::= PrintComp ( Expr+, )
   ```

2. Case 表达式的支持。
   新增 case 表达式（新增关键字 case 和 default），形如
   ```
   case(表达式) {
       常量1: 表达式1;
       常量2: 表达式2;
       …
       常量n: 表达式n;
       default: 表达式n+1;
   }
   ```
   其语义解释与 C 语言的 switch-case 控制结构相类似，不同之处只是表达式计算，而非执行语句。
   注：本学期的 case pattern（即上面的常量 1、常量 2、…、常量 n）仅限于整数类型的常量运算数。

参考语法：

```
Expr ::= case ( Expr ) { ACaseExor* DefaultExpr } | …
ACaseExor ::= Constant : Expr ;
DefaultExpr ::= default : Expr ;
```

3. 支持 super 表达式。
   类似于 this 表达式，但语义不同，祥见后续阶段的说明。
   参考语法：

```
Expr ::= super | …
```

4. 支持对象复制。
   新增表达式：深复制 dcopy(e) 和浅复制 scopy(e)，dcopy 和 scopy 为新增关键字，语义说明参见后续阶段。
   参考语法：

```
Expr ::= dcopy(Expr) | scopy (Expr) | …
```

5. 支持串行循环卫士语句。

   串行循环卫士语句的一般形式如

$$do\ E_1 : S_1 \ [\!]\ E_2 : S_2 \ [\!]\ \ldots \ [\!]\ E_n : S_n\ od$$

   我们将其语义解释为：

   (1) 依次判断布尔表达式 E_1，E_2，…，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（$1 \le k \le n$），则执行语句 S_k；转（1）。

   (3) 若 E_1，E_2，…，E_n 的计算结果均为 false，则跳出循环。

   本学期实验拟新增串行循环卫士语句（新增关键字 do 和 od）。

   参考语法：

```
Stmt ::= DoStmt ; | …
DoStmt ::= do DoBranch* DoSubStmt od
DoBranch ::= DoSubStmt [|]
DoSubStmt ::= Expr : Stmt
```

后续阶段实现将在你本次实验的基础上实现上述语言特性，因此本阶段作为基础非常重要，请大家认真完成。

语法规则说明

在 Decaf 语言规范中给出的参考文法是使用扩展的 EBNF（扩展巴氏范式）描述的，但是 BYACC 并不直接接受 EBNF 方式描述的文法，因此需要首先把 EBNF 形式的参考文法改写为 BYACC 所接受的上下文无关文法。

在改写 EBNF 为上下文无关文法的时候需要注意一些习惯写法：

1、 如果需要使用递归产生式，首选左递归（原因可以从 BYACC Manual 中找到）。

2、 形如 Stmt*这样的部分，我们将改写为：

StmtList -〉 StmtList Stmt | ε

并且用 StmtList 替代原文法中的所有 Stmt*的出现；形如 ClassDef+这样的部分，则改写为：

ClassDefList -〉 ClassDefList ClassDef | ClassDef

3、形如⟨Expr .⟩这样的部分，我们将改写为：

Receiver -〉 Expr . | ε

并且用 Receiver 替代原文法中的所有 Expr . 的出现。

我们已经在 parser.y 的模板中为大家定义好了终结符的代表码以及其文法名字（单字符操作符的代表码是其 ASCII 码，文法名字就是字符本身，例如′+′的文法名字就是′+′，注意有单引号）。在词法分析器中大家只需要识别出对应单词以后返回预先定义好的代表码即可（例如==返回的是 EQUAL）。

另外必须注意的是 parser.y 模板中示例性地给出 Decaf 语言规范中所有操作符的优先级和结合性说明，大家应当参照性的实现对本次实验新增语法特性，并体会指定算符优先级和结合性对消除冲突的作用。

关于详细的语法说明，请参考 Decaf 语言规范的参考文法一节。

抽象语法树（AST）

所谓的抽象语法树（Abstract Syntax Tree），是指一种只跟我们关心的内容有关的语法树表示形式。抽象语法是相对于具体语法而言的，所谓具体语法是指针对字符串形式的语法规则，而且这样的语法规则没有二义性，适合于指导语法分析过程。抽象语法树是一种非常接近源代码的中间表示，它的特点是：

1、 不含我们不关心的终结符，例如逗号等（实际上只含标识符、常量等终结符）。

2、 不具体体现语法分析的细节步骤，例如对于 List -> List E|E 这样的规则，按照语法分析的细节步骤来记录的话应该是一棵二叉树，但是在抽象语法树中我们只需要表示成一个链表，这样更便于后续处理。

3、 可能在结构上含有二义性，例如加法表达式在抽象语法中可能是 Expr -> Expr + Expr，但是这种二义性对抽象语法树而言是无害的——因为我们已经有语法树了。

4、 体现源程序的语法结构。

使用抽象语法树表示程序的最大好处是把语法分析结果保存下来，后面可以反复利用。

在面向对象的语言中描述抽象语法树是非常简单的：我们只需要为每种非终结符创建一个类。如果存在 A -> B 的规则的话我们就让 B 是 A 的子类（具体实现的时候考虑后面的处理可能有所不同）。

在我们的代码框架中我们已经为你定义好各种符号在 AST 中对应的数据结构。请在动手实现之前大致了解一下 decaf.tree.Tree 中所包含的各个类。你要在此文件中定义实验新增特性的语法节点。

提示

1、我们建议你按照由易到难的顺序，逐步增加词法分析程序识别的单词符号的类别。每增加一类，用 Lexer 的 diagnose 函数测试一下。

2、词法分析部分的重点是掌握 Flex 工具的用法，尤其是要掌握正规式的写法。以下是关于写正规式的几点建议：

- 为避免与 LEX 操作符混淆，最好对仅含有字符串常量的表达式都用""括起来。
- 有时使用宏可以使正规式更加简洁清晰。
- 可以通过 yytext()函数获得当前匹配到的字符串。

3、语法分析器部分的任务比较简单，大家只要把 Decaf 的 EBNF 语法规范改写成 BYACC 能够识别的语法规则并且在相应的归约动作中加入建立 AST 的语句就可以了。需要注意的是：Decaf 语言规范中给出的表达式规则有二义性，YACC 在处理时会出现冲突。有两种办法可以去掉冲突：一种是通过改写文法去掉二义性，例如 PL/0 语言表达式文法；另一种是利用优先级和结合性去掉二义性。在本实验中我们采用第二种方法，目的是希望大家熟悉 YACC 解决冲突的机制并体会这样做的好处。

4、构造语法分析器的过程中，有一个重要的问题是如何避免冲突。对此，我们有以下几点建议：

- 不要一次把所有的语法规则都添加进去，比较好的方法是逐次添加少量规则，并及时测试，这样不会导致冲突“大爆发”。
- 将左边符号相同的规则合并在一起，这样有利于避免手误。
- 各条规则的右部尽量对齐，这样比较清晰且易修改。
- 如果语法规则需要递归，尽量使用左递归，这样可以使语法分析程序尽早进行归约，不致使状态栈溢出。
- 在调试过程中，可以利用%start 子句来指定某个非终结符为开始符号，这样可以将注意力集中在部分规则上。
- 如果希望看到详细的冲突情况的报告，可以查看 parser.output 文件，里面详细报告了冲突的情况和 BYACC 选用的解决冲突方法。注意可能 YACC 默认的处理方法就是符合 Decaf 语言规范的，但我们仍然要求使用优先级和结合性来明确的消除冲突。

5、要充分考虑到输入的各种可能情况，我们给的例子只能涵盖一部分合法和不合法的输入。请自己设计一些例子，对程序进行全面的测试。测试也是实验的重要内容，对于自行设计的测例，若与已有测例的测试目标有本质不同，请在实验报告中指出，并附加该测例源码。

6、Decaf 每阶段实验所占最终成绩的比例见课程第一讲课件。程序部分主要看输出结果与标准输出的一致程度，占每阶段成绩的 80%。特别要注意，由于采用机器比对的方式评阅，因此你提交的输入输出内容和格式一定要与已给所有测例（见 TestCases\ S1 目录）完全匹配。报告部分主要看所提交的作业报告的描述，例如是否清楚说明了自己的工作内容等，占每阶段成绩的 20%。