Decaf 语言规范

在本课程中，我们将为 Decaf 语言编写一个编译器。Decaf 是一种强类型的、面向对象的、支持单继承和对象封装的语言。这一语言与 C/C++/Java 非常类似，因此你会发现很容易弄懂它。但是，它并不和这些语言中的任何一个完全一样。为了使这个作业的难度能够被接受，这里所采用的 Decaf 语言的特性是经过删减和简化的，但即使这样，你将会发现这种语言的表达能力仍然足够编写出各种漂亮的面向对象程序来。

这份文档给出本课程中 Decaf 语言的语法和语义规范，你在完成这个课程项目的过程中将会反复查阅。

词法规范

下面是 Decaf 的关键字，他们都是保留字：

bool break class else extends for if int new null return string this void while static Print ReadInteger ReadLine instanceof

一个标识符是以字母开头的字母、数字和下划线的序列。Decaf 是大小写敏感的，例如 **if** 是一个关键字，但是 IF 却是一个标识符，binky 和 Binky 是两个不同的标识符。

空白字符（即空格、制表符和换行符）仅用于分隔单词。关键字和标识符必须被空白字符或者一个既不是关键字也不是标识符的单词隔开。ifintthis 是单个标识符而不是三个关键字，但 **if**(23**this** 被识别成四个单词。

布尔常量是 **true** 或者 **false**，如同关键字一样，它们也是保留字。

一个整型常量既可以是十进制整数也可以是十六进制整数。一个十进制整数是一个十进制数字（0-9）的序列；十六进制整数必须以 0X 或者 0x 开头（是零，而不是字母 O），后面跟着一个十六进制数字的序列。十六进制数字包括了十进制数字和从 a 到 f 的六个字母（大小写均可）。合法的整数的例子有：8, 012, 0x0, 0X12aE。

一个字符串常量是被一对双引号包围的可打印 ASCII 字符序列。字符串常量中不可以包含换行符，也不可以分成若干行，例如：

```
"this is not a
valid string constant"
```

字符串常量中支持以下几种转义序列：\"表示双引号，\\表示单个反斜杠，\t 表示制表符，\n 表示换行符。其它情况下不认为反斜杠（\）是转义符。例如："\t"是一个长度为 1 的字符串，其中的\t 转义为制表符；而"\u"是一个长度为 2 的字符串，它包含反斜杠和字母 u 两个字符。

该语言中的操作符和分隔字符包括：

```
+ - * / % < <= > >= = == != && || ! ; , . [ ] ( ) { }
```

单行注释是以//开头直到该行的结尾。Decaf 中没有多行注释。如果单行注释出现在程序末尾，那么单行注释的结尾需要换行。

参考语法

参考语法以 EBNF 的扩展形式给出，将用到下列元符号：

x（粗体）	表示 x 是一个终结符（即“单词”）。 除个别关键字以外，本节中的终结符名字均为小写字母。
y（常规）	表示 y 是一个非终结符。非终结符的名字均为首字母大写。
〈x〉	表示 0 或 1 个 x 的出现，也就是说，x 是可选的。
x^*	表示 0、1 或多个 x 的出现。
x^+,	表示一个或多个以逗号分隔的 x
|	表示并列关系
ε	表示没有，即不存在任何符号

为了可读性起见，我们用操作符的词法形式表示它们，例如用!=而不是 NOT_EQUAL 等由词法分析器返回的代表码来表示不等号。

```
Program         ::= ClassDef +
VariableDef     ::= Variable ;
Variable        ::= Type identifier
Type            ::= int | bool | string |
                    void | class identifier | Type [ ]
Formals         ::= Variable+, | ε
FunctionDef     ::= 〈static〉Type identifier ( Formals ) StmtBlock
ClassDef        ::= class identifier 〈extends identifier〉 { Field* }
Field           ::= VariableDef | FunctionDef
StmtBlock       ::= { Stmt* }
Stmt            ::= VariableDef | SimpleStmt ; | IfStmt |
                    WhileStmt | ForStmt | BreakStmt ; | ReturnStmt ;|
                    PrintStmt ; | StmtBlock
SimpleStmt      ::= LValue = Expr | Call | ε
LValue          ::= 〈Expr.〉 identifier | Expr [ Expr ]
Call            ::= 〈Expr.〉 identifier ( Actuals )
Actuals         ::= Expr+, | ε
ForStmt         ::= for ( SimpleStmt ; BoolExpr ; SimpleStmt ) Stmt
WhileStmt       ::= while ( BoolExpr ) Stmt
```

```
IfStmt          ::= if ( BoolExpr ) Stmt ⟨else Stmt⟩
ReturnStmt      ::= return | return Expr
BreakStmt       ::= break
PrintStmt       ::= Print ( Expr+, )
BoolExpr        ::= Expr
Expr            ::= Constant | LValue | this | Call | ( Expr ) |
                    Expr + Expr | Expr - Expr | Expr * Expr |
                    Expr / Expr | Expr % Expr | - Expr |
                    Expr < Expr | Expr <= Expr | Expr > Expr |
                    Expr >= Expr | Expr == Expr | Expr != Expr |
                    Expr && Expr | Expr || Expr | ! Expr |
                    ReadInteger ( ) | ReadLine ( ) |
                    new identifier ( ) | new Type [ Expr ] |
                    instanceof ( Expr , identifier ) |
                    ( class identifier ) Expr
Constant        ::= intConstant | boolConstant |
                    stringConstant | null
```

程序结构

一个 Decaf 程序是一个类定义的序列，其中每个类定义包含该类的完整描述（也就是说 Decaf 中没有像 C++中的前视声明，实际上，Decaf 根本不需要这样的前视声明）。

一个 Decaf 程序应当包含一个名为“Main”的主类，主类中应包含一个名为 main、不带任何参数且返回类型为 void 的 static 方法。注意，从父类继承的 main 方法在这里不起作用。

作用域

Decaf 支持多种层次的作用域。最高层是全局作用域，其中只包含类定义。每个类定义有自己的类作用域。每个函数有一个用于声明参数表的参数作用域和存放函数体的局部作用域。局部作用域中一对大括号建立了一个嵌套的局部作用域。内层作用域屏蔽外层作用域。

- 类，函数和类成员变量可以在声明之前使用，唯一的条件是该符号是在引用处是可访问的。
- 局部作用域中的变量必须先声明后使用。
- 同一个作用域中的标识符是唯一的（Decaf 不支持函数重载）。
- 在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，但不允许在局部作用域中声明与外层的局部作用域或参数作用域中的变量同名的变量。另外，类名不会被任何非全局标识符屏蔽。
- 不可访问在一个已经关闭的作用域中声明的标识符。

类型

预定义好的基本类型有 **int**，**bool**，**string**，和 **void**。Decaf 允许类类型。数组类型可以通过任何非 **void** 的元素类型建立起来，支持数组的数组（AoA）。

变量

变量的类型可以是除 **void** 以外的基本类型、数组类型或者类类型之一。在一个类定义内部、但是不在任何函数内声明的变量具有类作用域。在函数参数表中声明的变量具有参数作用域，而在函数体中声明的变量具有局部作用域。一定被声明，则该变量保持可见直到该作用域关闭。

- 局部变量可以在语句序列的任意地方声明，而且在声明点到该声明所在的作用域末之间的区域可访问。

数组

Decaf 的数组是同构的（即数组每个元素都是同一种类型）线性索引的容器。数组的访问是通过引用的方式来实现的。数组的声明不包括大小信息，而且所有的数组都是在堆中使用内置操作符 **new** 按照所需的大小来动态分配的。

- 数组的元素类型可以是除 **void** 以外的任何基本类型、类类型或者数组（包括数组的数组）。
- **new** type**[**N**]**按照指定的元素类型和元素个数分配一个新的数组，这里 N 必须是非负整数，试图分配一个负长度的数组将会引起一个运行时错误。
- 数组的元素个数在分配数组的时候被记录下来，而且一旦分配完毕就不能再更改。
- 数组支持这样的特定语法类获取其元素个数： arr.length()。
- 数组索引的访问方式（即 arr[i]这样）只能够用在数组类型的变量上。
- 当索引一个超出数组范围的位置的时候将会出现运行时错误。
- 数组可以作为函数参数或者函数的返回值进行传递。数组对象本身通过传值方式进行传递，但是它是以引用方式来访问的，因此对数组元素的修改会反映到函数的调用方处。
- 数组赋值是浅拷贝（也就是说仅仅复制对数组空间的引用）。
- 数组比较（==和!=）仅比较引用是否相同。

字符串

Decaf 中的字符串支持很少。一个 Decaf 程序可以包含字符串常量、通过内置函数 **ReadLine** 从用户那里读取字符串，比较字符串，和打印字符串，但就只有这些了。Decaf 不支持程序创建和修改字符串，或者在字符串和其它数据类型之间进行转换等（可以考虑作为扩展）。字符串的访问是通过引用（指向字符串首址的指针）来实现的。

- **ReadLine**()读取用户输入的一行字符，直到换行符为止（但不包括换行符）。
- 字符串赋值是浅拷贝（也就是说，把一个字符串赋值给另一个字符串仅仅复制对字符串内容的引用）。
- 字符串可以作为函数的参数或者返回值进行传递。
- 字符串比较（==和!=）以区分大小写的方式比较两个字符串的字符序列。

函数的定义

函数的定义用于建立函数名字以及与这个名字相关联的类型签名，类型签名包括函数是否是静态的、返回值类型、形参表的大小以及各形参的类型。函数的定义提供类型签名以及组成函数体的语句。

- 函数必须定义在一个类作用域中，函数之间不允许嵌套。
- 函数可以有零或者多个形参。
- 形参的类型可以是除 **void** 以外的基本类型、数组类型或者类类型。
- 用在形参表中的标识符必须唯一（即形参不能重名）。
- 函数的形参是声明在函数体关联的局部作用域之外的另一个作用域中。
- 函数的返回类型可以是任何的基本类型、数组类型或者类类型。**void** 类型用于指出函数没有返回值。
- 一个函数只能被定义一次。
- 不支持函数的重载（overload），函数重载是指使用名字相同但类型签名不同的函数。
- 如果一个函数具有不是 **void** 的返回值类型，则其中的任何 **return** 语句必须返回一个与该返回类型兼容的值。
- 如果一个函数的返回值类型为 **void**，则它指能够使用不带参数的 **return** 语句。

函数调用

函数调用包括从调用方到被调用方传递参数值、执行被调用方的函数体、并返回到调用方（很可能带有返回值）的过程。当一个函数被调用的时候，要传递给他的实参将会被求值并且与对应形参进行绑定。Decaf 中所有的参数和返回值都通过传值的方式进行传递。

- 所调用的函数必须是有定义的，无论其定义是否出现在调用处之前。
- 函数调用中实参的个数必须与函数所需形参的个数相匹配。
- 函数调用中每个实参的类型必须与对应形参的类型相匹配。
- 函数调用时实参的求值顺序是从左至右。
- 函数调用过程中执行到一个 **return** 语句或者到达函数在源程序中的结尾时把控制权交还给调用方。
- 函数调用结果的类型是函数声明时候的返回值类型。

类

Decaf 程序中定义一个类的时候将会创建一个新的类型名称以及一个类作用域。一个类定义是一个成员域的列表，每一个成员域要么是一个变量，要么是一个函数——这些变量有的时候被称为实例变量、成员数据或者属性；这些函数被称为方法或者成员函数。

Decaf 通过一种简单的机制来强制对象的封装：所有的变量都是私有的（访问范围限于定义它的类及其子类，C++中称这种访问级别为 protected），所有的方法都是公开的（到处都可以访问）。因此，访问一个对象的状态的唯一手段是通过它的成员函数。

- 所有的类定义都是全局的，也就是说，类定义不能出现在函数中。
- 所有的类必须拥有唯一的名字。
- 一个成员域的名字在同一个类作用域中只能够使用一次（即不允许方法同名、变量同名或者变量和方法之间同名）。
- 成员域可以先使用后声明。
- 实例变量的类型可以是除 **void** 以外的基本类型、数组类型或者类类型。
- 在非静态方法中访问同一个类的成员域的时候“**this.**”的使用是可选的。

对象

一个变量如果其类型为类类型的话则称为对象，或者该类的实例。对象的访问以引用方式实现。所有的对象都是使用内置的 **new** 操作符在堆中动态分配的。

- 声明一个对象变量的时候所使用的类名必须有定义。
- **new** 参数中的类名必须是有定义的。
- 操作符.用于访问一个对象的成员域（变量或者方法）。
- 对于形如 expr.method() 这样的方法调用，expr 的类型必须是某个类 T，method 必须是 T 的成员方法的名字。对于静态方法，expr 可以是一个类名，对于非静态方法，expr 必须是一个对象。
- 对于形如 expr.var 这样的变量访问，expr 的类型必须是某各类 T，var 必须是 T 的成员变量的名字，而且这样的访问只能出现在类 T 或者其子类的作用域中。
- 对上一条的补充：在类作用域中，你可以通过该类或其子类的任何实例访问该类的私有变量，但不可以访问与该类无关的其他类实例的变量。
- 对象的赋值是浅拷贝（也就是说对象的赋值仅仅复制引用）。
- 对象可以作为函数的参数或者返回值进行传递。对象本身是通过传值的方式进行传递的，但是它通过引用的方式进行访问，因此对其成员函数的更改会反映到调用方那里。

继承

Decaf 支持单继承，允许子类通过添加或者覆盖已有方法来扩展基类。A 继承 B 的语义是 A 包含了在 B 中有定义的所有成员域（包括变量和函数），而且还有 A 自己的成员域。

子类可以覆盖（即通过重新定义的方式进行替换）一个继承而来的方法，但是重新定义的版本必须和原来的方法在返回类型和参数类型上一致。Decaf 支持自动类型提升（up-casting），因此一个类的对象可以用在任何需要使用其父类对象的地方。

所有的 Decaf 非静态方法都是动态绑定的（即 C++中的 **virtual**）。编译器在编译的时候不可能决定要调用方法的具体地址（考虑通过一个父类对象调用一个被子类覆盖了的方法），因此，函数地址的绑定是在运行时通过查询一张与各对象关联的成员方法列表（即虚函数表）来实现的，我们会在后面详细讨论它。

- 如果指定了父类，则父类的类型必须是有定义的类类型。
- 子类继承父类所有的成员域（包括变量和方法）。
- 子类不能覆盖继承而来的变量和静态方法。
- 子类能够覆盖继承而来的非静态方法（通过重新定义这个方法实现），但是新的版本的返回类型和参数类型必须和原有方法匹配。
- 一个类不能以不同的类型签名来指定两个同名的方法。
- 子类类型兼容于父类类型，子类的对象可以用于替换父类类型的表达式（例如，如果一个变量被声明为 **class** Animal 类型，而且 Cow 是 Animal 的一个子类，你就可以用一个 **class** Cow 类型的表达式给它赋值。类似地，如果 Binky 函数有一个类型为 **class** Animal 的形参，则可以传第一个 **class** Cow 类型的变量作为实参，对于返回 Animal 对象的函数，你也可以返回一个 Cow 对象作为返回值）。但反过来不成立（父类对象不能用来替换子类类型的表达式）。

- 前面的规则也适用于多级继承的情况。
- 检查一个对象的成员域的时候，判断是什么类的成员时看的是编译时这个对象是什么类的对象（也就是说，一旦你把 `Cow` 的实例提升成为 **`class`** `Animal` 类型的变量，则你不能通过这个变量访问那些 `Cow` 特有的成员域）。
- 数组没有子类或者类型提升之说：如果 `T2` 继承自 `T1`，则 `T2[]`类型的数组不兼容于 `T1[]` 类型的数组（这跟 Java 不同）。

反射

Decaf 支持 instanceof，可以通过 instanceof 来判断一个表达式的结果是否是某个类类型的实例。

- 用于 `instanceof` 参数表达式的结果必须是一个对象，而不能是某种基本类型的实例或者数组。
- 一个对象，除了是自己本身类类型的一个实例外，也是本身类类型的父类类型的实例（例如 `A` 是 `B` 的子类，那么一个 `A` 的实例 `a`，也是 `B` 的实例）。

类型的等价与兼容

Decaf 基本上是一个强类型的语言：每个变量都有一个特定的类型与之关联，而且该变量只能够容纳属于那种类型范围的值。如果类型 `A` 等价于类型 `B`，则其中某一类型的表达式可以自由地替换另一个类型的表达式。两种基本类型当且仅当他们是同一个类型的时候等价。两种数组类型当且仅当他们的元素类型等价的时候等价（元素类型本身可能也是一种数组，因此这意味着这里用到了结构等价的递归定义）。两种类类型等价当且仅当他们是相同的类型（也就是说仅仅是名字等价，而不是结构等价）。

类型的兼容性是受到更多限制的单向关系。如果类型 `A` 兼容于类型 `B`，那么一个 `A` 类型的表达式可以用来替换一个 `B` 类型的表达式，但反过来不一定。等价的类型在两个方向上都是兼容的。子类兼容于父类，但反过来不成立。**`null`** 类型兼容于所有的类类型。对于其他基本类型，兼容和等价的意义相同（注意，null 并不兼容于 string）。诸如赋值、参数传递等操作不仅允许等价的类型，而且允许兼容的类型。

强制类型转换

Decaf 支持类类型之间的强制转换，并且只支持类类型之间的强制转换，基本类型和数组不能进行强制转换。

- 被转换的必须是一个对象，而不能是基本类型和数组。
- 被转换的对象必须是转换到的类类型的一个实例，否则出现一个运行时错误。
- 如果被转换对象的类类型兼容于被转换到的类类型，则转变为自动的类型转换。

注意一个对象在编译期可以知道的类类型和其实际的类类型可能是不一样的（这也是为什么会有虚函数）。比如 A 继承 B，那么在很多需要 A 类型实例的地方，都可以用 B 类型的实例来替代，这个时候你可以把这个“A 类型”的实例，强制转换成 B 类型的实例。

赋值

对于基本类型（除字符串），Decaf 使用值拷贝（value-copy）的语义：语句 `LValue = Expr`

将会复制计算 Expr 所得到的结果值到 LValue 所对应的内存位置。对于数组，对象和字符串，Decaf 使用引用拷贝（reference-copy）的语义：语句 LValue = Expr 会使得 LValue 中存放有一个对于 Expr 计算结果的引用（也就是说，这种赋值复制的是指向对象的指针而不是对象本身）。换句话说，数组、对象和字符串的赋值是浅拷贝，不是深拷贝。

- LValue 必须是可被赋值的变量位置。
- 一个赋值语句右边的类型必须兼容于左边的类型。
- **null** 只能够赋值给具有类类型的变量。
- 在函数体中给形参赋值是合法的，这样的赋值只在函数体范围内有效。

控制结构

Decaf 控制结构是基于 C 和 Java 的，因此有很多相似之处。

- 一个 **else** 子句总是和最近的一个未闭合的 **if** 语句关联。
- **if**, **while** 和 **for** 语句的测试部分表达式的类型必须是 bool。
- 一个 **break** 语句只能出现在 **while** 或者 **for** 的循环体中。
- **return** 语句的返回值的类型必须与所在函数的返回类型兼容。

表达式

为了简单起见，Decaf 不允许在表达式中进行类型混合和转换（例如把一个整数当成布尔值使用等等）。

- 常数的求值结果就是常数值本身（**true**，**false**，**null**，整数，字符串常量）。
- 双操作数算术运算（+，-，*和/）的两个操作数必须都是 **int** 类型的，结果的类型跟操作数类型一致。
- %的两个操作数必须都是 **int** 类型的，结果类型是 **int**。
- 负号的操作数必须是 **int** 类型的，结果类型与操作数类型一致。
- 双操作数逻辑关系运算（<，>，<=，>=）的两个操作数必须都是 **int** 结果类型是 **bool**。
- 二元判等操作符（!= ，==）的两个操作数必须是等价类型（例如两个 **int**，但请参考下面关于对象判等的例外情况），结果类型是 **bool**。
- 二元判等操作符的两个操作数也可以是两个对象或者一个对象和 **null**，两个对象的类型必须在至少一个方向上兼容，结果类型是 **bool**。
- 所有的逻辑运算操作符的操作数必须都为 **bool** 类型，结果类型是 **bool**。
- &&和||不使用布尔短路；在计算结果之前两个表达式都需要求值。
- 所有表达式的操作数都是从左到右求值。算符的优先级从高到低如下：

```
[ .           （数组索引和成员域选择）
! -           （逻辑非，一元负号）
* / %         （乘，除，求余）
+ -           （加，减）
< <= > >=     （关系运算）
== !=         （判等操作）
&&            （逻辑与）
```

	（逻辑或）
=	（赋值）

所有的二元算术操作符和二元逻辑操作符都是左结合的。赋值和关系操作符不结合（也就是说，你不能把具有同样优先级的这些操作符连着来用：`a < b >= c` 或者 `a = b = c` 都不合法，但是 `a < b == c` 是可以的）。括号可以用于覆盖固有的优先级和结合性。

标准库函数（StdLib Functions）

Decaf 有一个非常小的标准库，可以用于简单的 I/O 和内存分配。标准库函数包括 **`Print`**，**`ReadInteger`**，**`ReadLine`** 和 **`new`**。

- 传给 **`Print`** 语句的参数只能是 **`string`**，**`int`** 或者 **`bool`** 类型。
- 创建对象的时候 **`new`** 参数中的类名必须是有定义的。
- 分配数组的时候传给 **`new`** 的第一个参数必须不是 **`void`** 类型，而且第二个参数必须是整型
- 分配数组的时候 **`new`** 的返回类型是一个元素类型为 T 的数组，这里 T 是作为 **`new`** 的第一个参数的那个类型。
- **`ReadLine`**`()` 读取用户输入的一个字符序列，直到换行符为止（不包括换行符）。
- **`ReadInteger`**`()` 读入用户输入的一行文本，并把它转换成整数（如果用户输入不合法，返回 0）。

运行时检查

Decaf 仅支持三种运行时检查（这留下很大的扩展余地）：

- 数组的下标所指示的位置必须在数组范围内，也就是说，在 `0..arr.length()-1` 这个范围中。
- 分配数组的时候传给 **`new`** 的数组大小必须是非负的。
- 强制类型转换时，检查被转换的对象是否是要转换到的类类型的一个实例。

当发生运行时错误的时候，会在终端上输出一个合适的错误信息，然后程序终止。

Decaf 不能做的事情

Decaf 被设计为一个简单的语言。虽然它具备了所有编写各种面向对象程序所需要的特点，但是仍然有不少 C++和 Java 的编译器做到但它做不到的事情。这里是一些想到的例子：

- 不检查通过 **`null`** 对象访问方法或者成员变量。
- 不检测任何不可到达的代码。
- 没有释放内存的函数或者垃圾回收机制，因此动态分配的空间不会被回收。
- 不支持对象的构造函数和析构函数。
- 还有很多其他的……

ACKNOWLEDGEMENT

本语言规范最初源自蒋波同学的翻译版本，后经历过部分助教同学的修改补充，在此表示衷心的感谢。

参与过 Decaf 实验的助教：梁英毅，张铎，曹震，李叠，蒋挺宇，许建林，谢宇轩，唐硕，毛雁华，蒋波，张迎辉，刘天淼，高崇南，王耀，等等。由于统计遗漏，有一些同学未列出，在此一并致谢。杨俊峰校友在引入 Decaf 实验时提供了帮助，以及当前 Decaf 实验框架基于 Julie Zelenski 教授教学组的原始工作，并参考了 Alex Aiken 教授的工作，在此向他们深表感谢。