Decaf PA3 说明

任务描述

在 PA2 中，我们已经完成了对输入程序的语义分析工作，此时的输入程序必定是有明确的语义而且不具有不符合语言规范的语义错误的，在接下来的 PA3 中，我们将对该输入程序进行翻译，把使用带属性修饰的抽象语法树（decorated AST）来表示的输入程序翻译成适合后期处理的另一种中间表示方式。

一般来说，把程序翻译为 AST 以外的中间表示方式是各种编译器前端的最后一步工作，这步工作以后编译过程便进入了中端处理或者直接进入后端处理部分。为了便于翻译工作的进行，在实际的编译过程中往往需要反复地从一种中间表示变为另一种中间表示，直到变换成最终的汇编代码或者二进制目标代码为止，其中后一种中间表示总是更加适合后面所需要进行的分析和处理。有的时候为了更好地分析程序的需要中间表示之间的变换次数是非常多的，例如 GCC 4 的编译过程除 AST 以外还有 GENERIC、GIMPLE、RTL 几种中间表示；而 ORC/Open64 的编译过程除 AST 以外还有多达九种的中间表示。

在 Decaf 编译器中，为简单起见，我们在 AST 以外只涉及一种中间表示，这种中间表示叫做三地址码（Three Address Code，TAC），是一种比较接近汇编语言的中间表示。PA3 中我们需要把 AST 表示的程序翻译为跟它在语义上等价的 TAC 中间表示，并在合适的地方加入诸如检查数组访问越界、数组大小非法等运行时错误的内容。在 TAC 表示的基础上，在下一阶段（PA4）构造控制流图，进行数据流分析，进行中间代码优化，最后（PA5）生成汇编代码以后，整个编译过程便告完成。

通过这个阶段，希望大家能在语法制导处理的基础上进一步掌握语法制导的中间代码翻译方法，并且对过程调用约定、面向对象机制的实现方法、存储布局等内容有所了解。

本阶段涉及的类和工具说明

文件/类	含义	说明
translate/Translater	翻译工作的辅助类	根据需要修改
translate/TransPass1	第一趟扫描	根据需要修改
translate/TransPass2	第二趟扫描	根据需要修改
tac/*	TAC 语句	基本不需要修改，但不排除某些实现方案对其进行小规模改动
frontend/*	编译器最前端	你要用 PA1-A 中修改过的文件覆盖目录下相应的文件
typecheck/*	语义检查部分	你要用 PA2 中修改过的文件覆盖目录下相应的文件

scope/*	作用域定义	不需要修改
symbol/*	符号定义	不需要修改
type/*	类型定义	不需要修改
tree/*	抽象语法树的各种结点	你要将前两个阶段中修改过的部分复制过来，根据 PA3 需要可对其进行修改
error/*	表示编译错误的类	根据需要增加运行时错误类型。 前面阶段工作，请自行移植
backend/*	后端支持	根据需要修改
machdesc/*	机器描述	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	不需要修改，打包部分不要修改

在 PA3 中，编译器的输出是 TAC 代码。我们提供了一个 TAC 模拟器 TestCases/S3/tac.jar，可以解释执行编译器输出的 TAC 代码。执行 java -jar tac.jar [-m num] <tac 文件>，就可以运行一个 tac 文件了，-m num 是可选的参数，表明模拟器最多运行多少条指令之后退出，默认是 100000 条，这主要是为了防止 TAC 中有死循环。

代码框架中的 TAC 表示包括了数据对象表示和执行语句表示两块内容，其中的细节请参考背景知识中的相关说明。

代码框架中给出的翻译过程通过对 AST 的两趟扫描完成：第一趟扫描计算每个类对应的类对象的大小，处理类成员方法的顺序，创建虚函数表，计算类成员变量偏移量，为函数创建 Functy 对象，并计算形参的偏移量。第二趟扫描完成所有语句和表达式的翻译工作。为示例起见，代码框架中直接给出了第一趟和第二趟扫描的基本实现，即 decaf 语法规范中描述的，但不包含新增特性。

我们并不要求大家必须采用我们提供的框架和实现思路：大家可以自行设计自己的实现方式，甚至可以推翻 TransPass1 和 TransPass2 全部现有代码。但是不应修改 TAC 代码的格式（否则模拟器无法运行）。因此我们在评分时也不会对比输出的 TAC 代码，而是对比它在模拟器上运行的结果。

实验框架中包含少量运行时错误检查，比如针对数组的两种运行时错误：数组下标越界和数组大小非法。数组下标越界是指访问一个数组元素的时候数组下标不在 0 到 length()-1 的范围内，这个错误需要在访问一个数组元素时检查；数组大小非法是指创建一个数组的时候所指定的数组长度小于零，这个错误需要在分配数组空间时检查。（阅读 Translater.java 中的 genCheckArrayIndex 和 genCheckNewArraySize 函数。）

这一阶段有一个特殊的样例程序 blackjack.decaf。这个程序是二十一点游戏，它在运行时会读标准输入（键盘）。runAll.py 不会编译/运行它，大家可以在实现新增特性之前自行测试，测试方法：

```
java -jar decaf.jar -l 2 blackjack.decaf >blackjack.tac
```

（编译为 TAC 代码，并保存到 blackjack.tac 文件）

```
java -jar tac.jar blackjack.tac
```

（运行）

实验内容

在本次实验中，依旧给出满足《decaf 语言规范》的基本框架实现，要求和前几阶段一样，增加新语言特性的实现，对 AST 进行扫描，完成相应语句和表达式的翻译工作。开始时，你需要将前面阶段的工作复制到本次实验的框架中。建议你首先要充分理解基本框架的代码结构及功能，参考框架中对相似语言特征的处理过程，根据自己对语言的理解完成新增语言特性的 TAC 代码生成。

需要实现的新增语言特性如下：

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。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 imgConstant 表示复数常量：

   Constant ::= intConstant | boolConstant | imgConstant | …

   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。对于表达式 @e 和 $e，e 必须是 **complex** 类型的表达式，且这两个表达式计算结果的类型为 int。对于表达式 #e，e 必须是 int 类型的表达式，且该表达式计算结果的类型为 **complex**。

   参考语法：

   Expr ::= @ Expr | $ Expr | # Expr | …

   4）新增复数打印语句 PrintComp（E_1，E_2，…，E_n），表示复数表达式 E_1，E_2，…，E_n 计算结果的显示（表达式 E_1，E_2，…，E_n 均要求具有 **complex** 类型），输出结果应当为 a+bj（中间不含空格，即使 a,b 为 0 也输出即可以出现 0+0j 样式的输出）。

   参考语法：

   Stmt ::= PrintCompStmt ; | …

   PrintCompStmt ::= **PrintComp** ($Expr^+$,)

   5）本学期，我们限定复数表达式仅包含加法（+）和乘法（*）运算，即不支持含有其他运算的表达式。

   6）要求支持对复数类型变量的赋值，此时，用于赋值的表达式要求具有 **complex** 类型。

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。对于 case 表达式：

```
case(E) {
    C1: E1;
    C2: E2;
    …
    Cn: En;
    default: En+1;
}
```

1）E 必须是 **int** 类型的表达式。

2）C_1、C_2: C_2、…、C_n 是互不相同的int类型常量。

3）E_1、E_2: E_2、…、E_n 以及 E_{n+1} 是具有相同类型的表达式。

4）该表达式计算结果的类型与E_1、E_2: E_2、…、E_n 以及 E_{n+1} 具有相同的类型。

参考语法：

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

3. 支持 super 表达式。

1）类似于 this 表达式，super 表达式返回当前对象，其类型为当前对象的 class。

2）本次实验中，我们限定仅支持面向 super 的函数调用（call 表达式），而不支持面向 super 的成员变量访问。

3）面向 super 的函数调用（call 表达式）super.f(…)，是调用当前对象的 class 的超类中的成员函数，即从其父类开始向上搜索类层次首次发现的成员函数 f(…)，如果未搜索到则报错。

4）本次实验测例中，不会出现 super 单独出现在等号右边的情况。

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)：

1） 对于新增表达式深复制 dcopy(e) 和浅复制 scopy(e)，e 必须是 **class** 类型的表达式，且这两个表达式计算结果将生成新的对象，该对象的类型为与 e 相同的 **class 类型**。

2） 当表达式 dcopy(e) 和 scopy(e) 出现在赋值语句中时，比如 x=dcopy(e)、x=scopy(e)，被复制的变量 x 须具有与 e 相同的 class 类型。

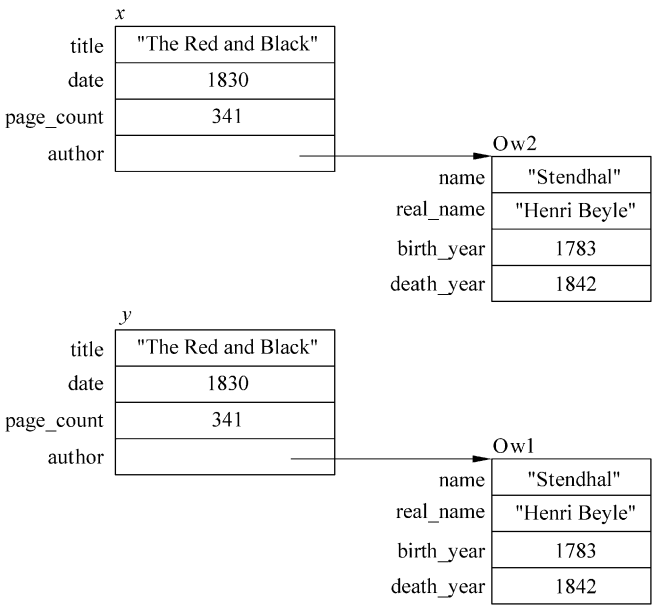
参考语法：

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

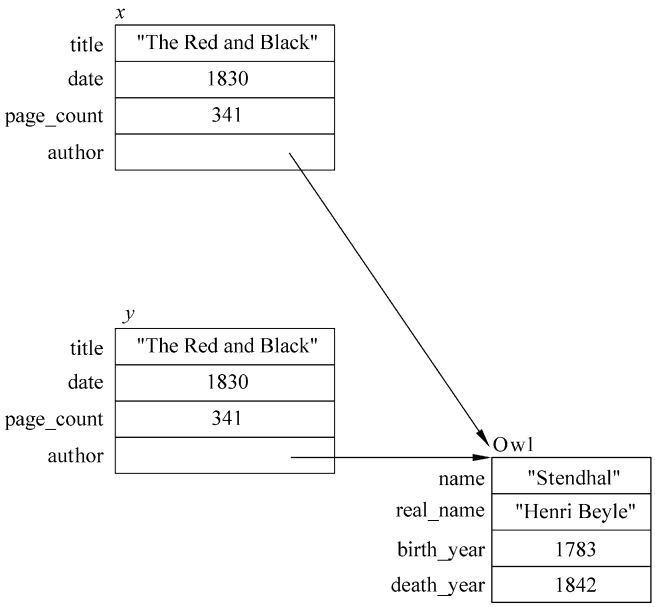
以下进一步解释本实验中表达式 dcopy(e) 和 scopy(e)的语义（注：所有表示对象的图为高层设计图，均不对应底层实现时的存储组织）：

设有“书”和“作者”两个类。

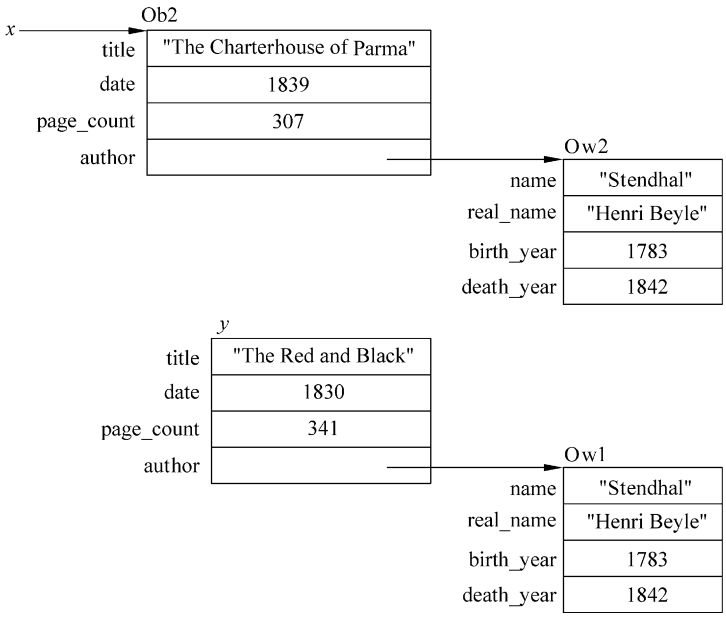
如下图所示，y 是一个“书”对象，Ow1 是一个“作者”对象。那么，“书”对象 x 可以认为是 dcopy(y)的一个结果对象，注意 Ow2 是不同于 Ow1 的另一个“作者”对象。



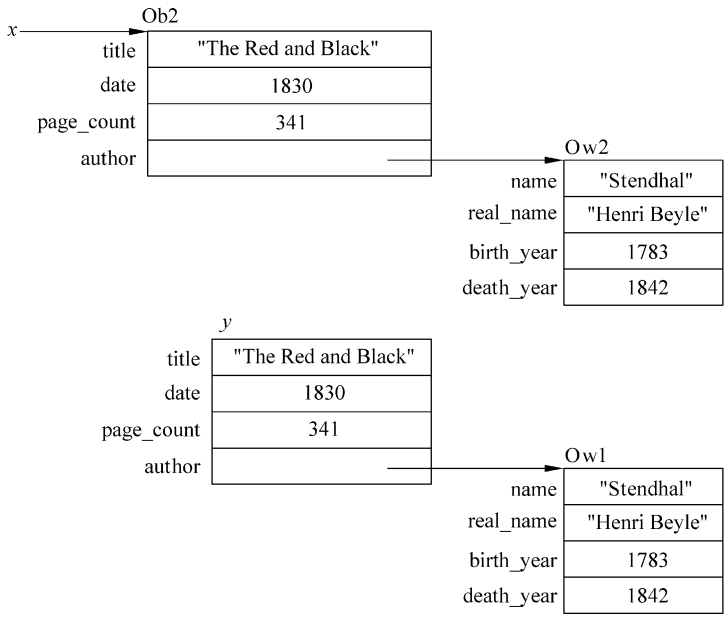
如下图所示，y 是一个“书”对象，Ow1 是一个“作者”对象。那么，“书”对象 x 可以认为是 scopy(y)的一个结果对象，注意对象 x 与对象 y 共享作者对象 Ow1。



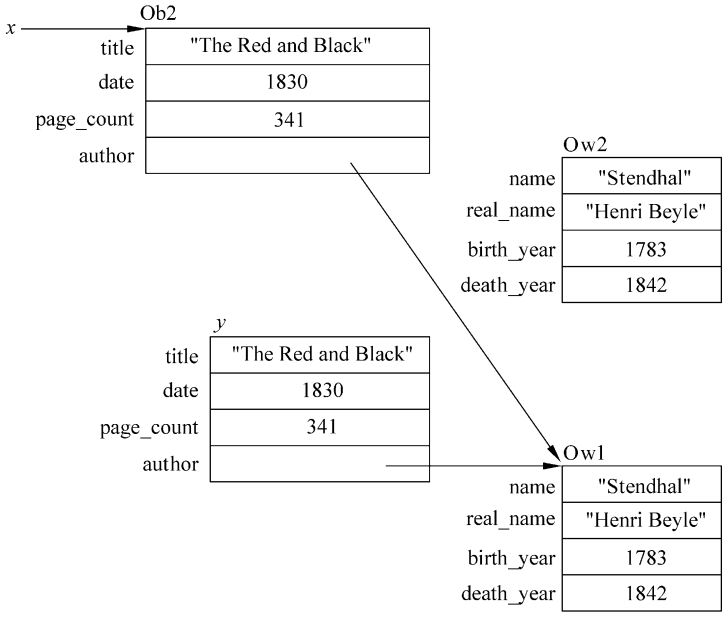
下面进一步解释一下赋值语句 x=dcopy(y) 和 x=scopy(y) 的含义，其中 y 与上面一样，是一个“书”对象，而 x 则是一个类型为“书”的变量。在赋值前，x 可以有一个对象值，也可以没有初始化。赋值以后结果都拥有了一个新的对象值。我们假设是前者，x 已经有一个对象值。设下图是赋值前的情形：



下图描述了在执行赋值语句 x=dcopy(y) 之后的情形：



下图描述了在执行赋值语句 x=scopy(y) 之后的情形：



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）。

对于串行循环卫士语句：

[1] E_1 ， E_2 ， …， E_n 必须是 bool 类型的表达式。
[2] 其静态语义规则类似于框架中已有的其他循环语句的规则。
[3] 语义错误的处理可可参照框架中已有的其他循环语句的处理方法。
[4] 串行循环卫士语句中支持 break 用法，其语义是直接跳出串行循环卫士语句。

参考语法：

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

本次实验将不针对新特性引入新的 tac 语句，在给新特性产生 tac 码时，应设法根据已有的 tac 语法进行设计。

本阶段要求检测一种运行时错误：“除零”非法。可参考框架中的两种关于数组的运行时错误检测：数组下标越界和数组大小非法。

截止时间

PA3 持续时间为 18 天，提交时间以网络学堂为准。

本阶段的测试要求仍然是输出与标准输出完全一致（忽略空白字符和换行符）。如前面的阶段，这一阶段也将包含一些隐藏的测试样例，因此你需要在调试的时候自己设计一些例子进行测试，看看运行结果是否合理。

Decaf 中的 TAC 简介

中间代码的目的是为了更好地对要翻译的内容进行加工处理，例如对代码进行优化等。中间代码的种类有多种，例如逆波兰表达式、三元式、TAC（三地址码，即四元式）、DAG

（有向无环图）、类 LISP 表达式等等。不同的中间代码有不同的适用场合，例如有些适合进行代码逻辑分析，有些适合用来进行源语言层次的优化，有些适合用来做机器层次的优化等等，因此同一个编译器内可能会使用多种中间代码（源代码->中间代码 1->中间代码 2->…->目标代码）。在 Decaf 项目中我们只使用 TAC 作为我们的编译器中间代码，每条 TAC 最多可以有三个参数，例如加法指令中含有两个操作数地址和一个结果地址（所谓地址这里是指临时变量）。我们采用的是“源代码->AST->TAC->MIPS 代码”的翻译过程。

我们在实验框架中的 tac 包下里面已经定义好了需要用到的所有的 TAC 种类（事实上你几乎不会直接使用它们，而是通过 Translater 类的方法来创建 TAC 实例）。

下面这张表是 Decaf 中使用的 TAC 简介，更加详细的内容请参考框架代码及注释。

名字	显示形式示例	含义
赋值操作		
ASSIGN	x = y	把变量y的值赋给变量x
LOAD_VTBL	x = VTBL <C>	把类C的虚表地址加载到x中
LOAD_IMM4	x = 34	加载整数 34 到变量x中
LOAD_STR_CONST	x = “hello world”	加载字符串常量到变量x中
算术运算操作		
ADD	c = (a + b)	把a和b的和放到c中
SUB	c = (a - b)	把a和b的差放到c中
MUL	c = (a * b)	把a和b的积放到c中
DIV	c = (a / b)	把a除以b的商放到c中
MOD	c = (a % b)	把a除以b的余数放到c中
NEG	c = -a	把a的相反数放到c中
逻辑与关系运算操作		
EQU	c = (a == b)	若a等于b则c为 1，否则为 0
NEQ	c = (a != b)	若a不等于b则c为 1，否则为 0
LES	c = (a < b)	若a小于b则c为 1，否则为 0
LEQ	c = (a <= b)	若a小于等于b则c为 1，否则为 0
GTR	c = (a > b)	若a大于b则c为 1，否则为 0
GEQ	c = (a >= b)	若a大于等于b则c为 1，否则为 0
LAND	c = (a && b)	把a和b逻辑与操作的结果放到c
LOR	c = (a || b)	把a和b逻辑或操作的结果放到c
LNOT	c = !a	把a逻辑非的结果放到c
控制流管理		
BRANCH	branch _L2	无条件跳转到行号_L2 所表示的地址
BEQZ	if (c == 0) branch _L1	如果c为 0 则跳转到_L1 所表示地址
BNEZ	if (c != 0) branch _L1	
RETURN	return c	结束函数并把c的值作为返回值返回
函数调用相关操作		

PARM	parm a	变量a作为调用的参数传递
INDIRECT_CALL	x = call a	取出a中函数地址，并调用，结果放x
DIRECT_CALL	x = call _Alloc	调用运行时库函数_Alloc，结果放x
内存访问操作		
LOAD	x = *(y - 2)	把地址为y-2 的单元的内容加载到x
STORE	*(x + 4) = y	把y保存到地址为x+4 的内存单元中
其他		
MARK	_L5:	定义一个行号_L5（全局的）
MEMO	memo 'XXX'	注释：XXX（供模拟器使用）

TAC 表示中使用的数据对象如下

名字	含义
Temp	临时变量
Label	标号
Functy	函数块
VTable	类的虚函数表

其中 Temp 与实际机器中的寄存器相对应。在 Decaf 框架中，我们用临时变量来表示函数的形式参数（parameter/formal argument）以及函数的局部变量（但是不表示类的成员变量）。在 PA3 的第一趟 AST 扫描中，我们将会为所有函数的所有形式参数关联上对应的 Temp 对象，在第二趟扫描中遇到 VarDef 时为其关联 Temp 对象。在后面的 AST 扫描过程中可以通过 Variable 的 getTemp()函数来获得所关联的 Temp 对象。此外，在翻译的过程中还可以通过 Temp 的 createTempI4()函数获取一个新的表示 32 位整数的临时变量。与实际寄存器不同的是，一个 Decaf 程序中可以使用的临时变量的个数是无限的。

Label 表示标号，即代码序列中的特定位置（也称为“行号”）。在 Decaf 框架中有两种标号，一种是函数的入口标号，另一种是一般的跳转目标标号。正如我们在前面介绍，TAC 是一种比较接近汇编语言的中间表示，因此诸如分支语句、循环语句等等将会转换成在一系列行号之中进行跳转的操作（即有些语言中的 GOTO 语句）。在 PA3 中，我们在第一趟 AST 扫描的时候为各 Function 对象创建 Functy 对象，其中就有函数的入口行号信息，在后面的 AST 扫描过程中可以用 Function 的 getFuncty()函数来获得函数的函数块然后得到入口行号对象。

Temp 和 Label 都是用于函数体内的数据对象，在 Decaf 框架的 TAC 表示中，我们用 Functy 对象来表示源程序中的一个函数定义。与符号表中的 Function 对象不同，Functy 对象并不包括函数的返回值、参数表等等信息，而仅仅包括了函数的入口标号以及函数体的语句序列。

VTable 所表示的是一个类的虚函数表，即一个存放着各虚函数入口标号的数组。关于虚函数表的细节请参看后面的相应章节。

最后，一个 VTable 列表加上一个 Functy 列表，就组成了一个完整的 TAC 程序。

从上面对 TAC 中间表示的描述可以看出，该中间表示是一种比 AST 更低级、但比汇编

代码高级的表示方式（具有“函数”、“虚函数表”等概念）。

在我们给出的 Decaf 代码框架中，已经为大家把创建各种 TAC 的方法封装成 Translater 类，大家需要利用该类建立各种数据对象、并且对函数体的各种语句和表达式进行翻译。需要注意的是，在开始翻译函数体之前需要调用 Translater 的 beginFunc()函数来开始函数体的翻译过程，在翻译完函数体以后需要调用 Translater 的 endFunc()函数来结束函数体的翻译过程（否则将不能形成正确的 Functy 对象）。

Decaf 提供的运行时库函数说明

一般来说，编译器在把源程序转换为目标机器的汇编程序或者机器代码的过程中，除了直接生成机器指令来实现一些功能以外，有时还会调用运行时库函数所提供的功能。所谓的运行时库（runtime library），是指一系列预先实现好的函数的集合（请注意跟 Decaf 语言规范中的“标准库函数”不同），这些函数往往是针对特定的运行平台实现的，帮助编程语言实现一些平台相关的功能，例如 C 语言中的 libc 库（在 windows 平台上通常是 MSVCRT.dll），又例如 Java 语言的类库（例如 rt.jar）等。通常，这样的运行库是随着所使用的编译器（或者解释器）的不同而不同的。

在 Decaf 中，为了实现一些平台相关的功能，我们也提供了一系列的运行时库函数，这些函数涉及到内存动态分配、输入输出等等功能。这些函数我们都定义在 machdesc.Intrinsic 中，通过 DirectCall 的方式来进行调用。注意库函数调用的传参方式和其他函数调用是一样的，也需要事先 push 参数。具体调用方法可以参考框架中给出的一些例子。

以下是对 Decaf 运行时库中所提供的八种运行时库函数的具体介绍：

- Intrinsic.ALLOCATE
  功能：分配内存，如果失败则自动退出程序
  参数个数：1
  参数 1：为要分配的内存块大小（单位为字节）
  返回：该内存块的首地址
- Intrinsic.READ_LINE
  功能：读取一行字符串（最大 63 个字符）
  参数个数：0
  返回：读到的字符串首地址
- Intrinsic.READ_INT
  功能：读取一个整数
  参数个数：0
  返回：读到的整数
- Intrinsic.STRING_EQUAL
  功能：比较两个字符串
  参数个数：2
  参数 1，2：要比较的两个字符串的首地址
  返回：若相等则返回 true，否则返回 false
- Intrinsic.PRINT_INT
  功能：打印一个整数

参数个数：1
参数 1：要打印的数字
返回：无

- Intrinsic.PRINT_STRING
  功能：打印一个字符串
  参数个数：1
  参数 1：要打印的字符串首地址
  返回：无
- Intrinsic. PRINT_BOOL
  功能：打印一个布尔值
  参数个数：1
  参数 1：要打印的布尔变量
  返回：无
- Intrinsic.HALT
  功能：结束程序，可以作为子程序调用，也可以直接 JUMP
  参数个数：0
  返回：无

运行时存储布局

一般来说，程序运行时的内存空间从逻辑上分为“代码区”和“数据区”两个主要部分。顾名思义，代码区用于存放可执行的代码，而数据区用于存放程序运行所需的数据（例如临时变量、虚函数表的空间等等）。

数据区按照所存放的数据和对应的管理方法分为全局数据区（静态数据区）、栈区、堆区三部分，这三个区里面的存储空间分配分别遵循三种不同的规则：静态存储分配（Static Memory Allocation）、栈式存储分配（Stack-based Allocation）和堆式存储分配（Heap-based Allocation），其中后两种分配方式称为“动态存储分配”，因为这两种方式中存储空间并不是在编译的时候静态分配好的，而是在运行时才进行的。

1、 全局数据区（静态数据区）

全局/静态数据区用于存放各种全局变量、静态变量还有类的虚函数表。静态存储分配的结果是编译的时候确定的，在进行编译的时候编译器根据全局变量等信息事先计算好所需的存储空间以及各变量在这个存储空间中的偏移地址。在 C 语言中全局数组的存储分配方法即为静态存储分配。

静态存储分配并不是总适用的，对于一些动态的数据结构，例如动态数组（C++中使用 new 关键字来分配内存）以及可重入函数的局部变量（例如 Hanoi Tower 问题中递归函数的局部变量）等最终空间大小必须在运行时才能确定的场合静态存储分配通常无能为力。

2、 栈区

栈区顾名思义就是作为“栈”这样一种数据结构来使用的。栈区数据空间的存储管理方式称为栈式存储分配。与静态存储分配方式不同，栈式存储分配是动态的，也就是说必须是运行的时候才能确定分配结果的，比方说以下一个计算阶乘的 C 代码片断：

```
int factorial (int n) {
    int tmp;
    if (n <= 1)
    return 1;
    else {
    tmp = n - 1;
    tmp = n * factorial(tmp);
    return tmp;
    }
}
```

这段代码中，随着 n 的不同，tmp 变量所需要的总内存空间大小是不同的，而且每次递归的时候 tmp 对应的内存单元都不同。诸如局部变量的栈式存储分配方法想必大家在学习 C++或者汇编语言的时候已经有所了解，函数调用时的存储布局情况，请参考后面章节。

分析函数调用时候的存储布局情况我们不难发现，进行栈式存储分配的条件是在编译的时候需要知道一个函数的活动记录有多大(以便在进入函数的时候动态地分配活动记录的空间)，如果这点不能满足，则应该使用堆式存储管理。

一般来说，栈区中的数据通常都是函数的活动记录，活动记录中的数据通常是使用寄存器偏址寻址方式进行访问的。所谓寄存器偏址寻址方式，即在一个基地址寄存器中存放着活动记录的首地址，在访问活动记录某一项内容的时候只需要使用该首地址以及该项内容相对这个首地址的偏移量即可计算出要访问的内容在内存中的实际逻辑地址。这类数据包括了函数的形式参数以及局部变量，具体细节请参考后续章节。

3、 堆区

堆是栈以外的另一种动态存储分配结构，它有两个基本的操作：申请内存和释放内存，C++的 new 和 delete 两个关键字即对应这两种功能。关于堆式管理在学术界和工业界都进行了广泛的研究和探索，有兴趣的同学可以参考 Wikipedia 的这个页面：

http://en.wikipedia.org/wiki/Dynamic_memory_allocation

堆式管理是应用程序在运行的时候通过向操作系统请求分配内存（例如 UNIX 中使用 sbrk 系统函数）和释放内存来实现的，因此分配和销毁都要占用相当的时间。在 Decaf 里面，数组和类对象都是在堆区上来分配内存空间的（因此需要用 ALLOCATE 运行时库函数）。

一般函数调用过程

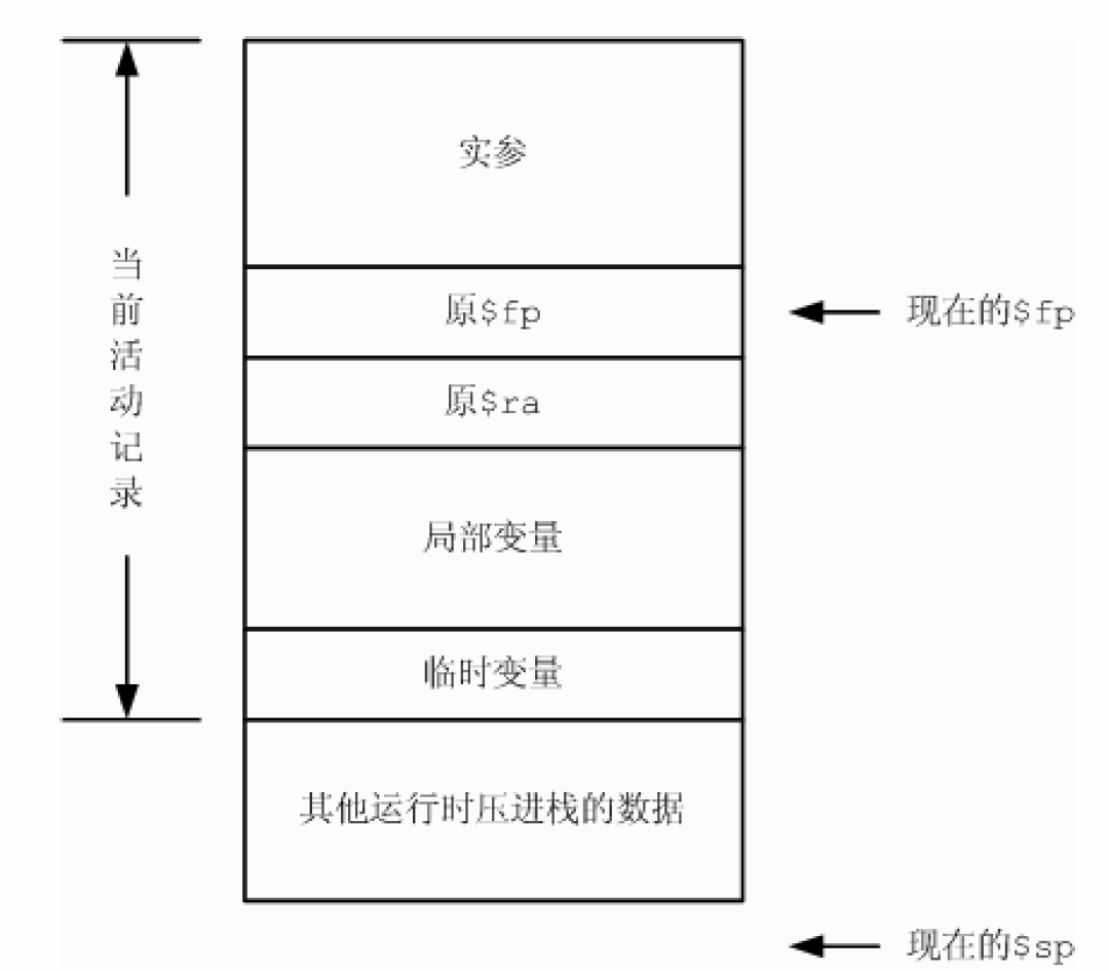
在这一节我们以 C 语言中的函数在 MIPS 机器上的调用过程为例（在 x86 上的过程类似），介绍一般函数的调用过程。对于 C++中的虚函数以及 Decaf 中的函数，其调用过程略有不同，但基本的原理是相同的。

C 语言程序一个典型的函数调用过程是这样的：

a) （调用方）把函数的实际参数按照参数表中从右到左的顺序压栈
b) （调用方）使用跳转语句跳转到函数入口（并把返回地址保存在$ra寄存器中）
c) （调用方或者被调用方）保存好返回地址
d) （被调用方）保存好原基地值寄存器$fp的值
e) （被调用方）把基地值寄存器$fp指向栈顶指针寄存器$sp所指的位置

f) （被调用方）修改栈顶指针寄存器的值$sp，使得$sp和$fp之间的区域足够存放函数的联系单元、局部变量和临时变量等
g) （被调用方）执行函数体代码直到函数返回
h) （被调用方）恢复原来的$sp和$fp
i) （被调用方）跳转回保存好的返回地址所指处
j) （调用方）释放之前压进栈的那些实参的空间（通过弹栈操作）

函数执行的时候的内存布局情况如图所示：



所谓活动记录（activity record），就是指当前函数执行所需的局部信息区域，这个区域包括了：传进函数的实参（在调用函数之前压进栈）、联系单元（动态链 2 和返回地址 3）、函数的局部变量和各种临时变量，在诸如PL/0 这样的支持函数嵌套定义的语言中，运行时联系单元中还会包括静态链信息。

在函数体中，访问函数的一个形式参数实际上是对实参区域进行访问，此时需要使用当前的基地值寄存器$fp加上该形参相对于当前栈帧首地址的偏移量进行访问，这是因为虽然在运行的时候存放着该形参内容的内存单元的地址在编译的时候是不可预先知道的，但是这个单元在活动记录中的位置在编译的时候是可以预知的。同样的访问方式也适合于对函数的局部变量以及临时变量的访问中。可见，在这种意义下，函数的参数、局部变量、临时变量三者并没有实质的区别，因此在Decaf代码框架中我们都使用Temp对象来表示这三种数据对象，其中唯一不同的是实参相对于当前栈帧首地址的偏移量是需要在PA3 中预先算好的，而局部变量和临时变量的偏移量都是在PA4 中进行数据流分析以后才确定(实际上可能使用

寄存器就已经足够了，不需要偏量）。

在这次的decaf tac描述中，我们使用PARM来表示参数的传递，因此在TAC层实际上隐藏了实际的入栈操作，我们只需要按照从左到右的顺序传递参数即可。需要注意的地方是，TAC模拟器在执行时会将PARM传递的参数当做这个PARM之后第一次碰到的CALL的参数。

各种数据对象的内存表示

为了简单起见，我们在Decaf语言中不对浮点数进行支持。并且我们约定：

- 对于int类型的常量和变量，对应于 32 位整数，即 4 占字节
- 对于bool类型的常量和变量，使用 32 位整数实现（通常是用 8 位整数实现），以 0 表示false，1 表示true
- 对于string类型的常量和变量，用字符串的首地址（32 位）的来表示字符串，字符串本身不记录长度，使用 0 字符（即'\0'）作为结束符，结束符不是字符串的一部分。这些都与C语言相同
- 对于数组，采用运行时动态分配的方式来分配存储空间，所分配到的存储区域大小为数组长度加 1 个单元，每个单元为 32 位（即 4 字节，以下同），其中第一个单元用于记录数组的长度（数组的length()函数用到这个信息），后面的单元均为数组内容，整个数组用指向第一个元素的指针（32 位地址）表示
- 对于类对象，也是采用运行时动态分配的方式来进行存储空间分配，所分配到的区域应该包含以下内容：第一个单元存放着这个对象所对应的类的虚函数表地址（Virtual Table，请回忆C++中关于虚表的概念），这个单元的内容在调用一个类的成员函数的时候起着很重要的作用；后面的单元均用于存放与这个对象相关联的各个成员变量，存储位置越是靠近第一个单元，则该成员变量的“辈分”越高，例如三个类A、B、C有B继承于A而且C又继承于B的关系，这三个类分别有成员变量a、b、c，则对于一个类C的对象来说，a、b、c三者中存储位置最靠近第一个单元的应该是成员变量a，其次是b，离第一个单元最远的应该是c；类对象用指向对应存储区域第一个单元的指针来表示（更详细的说明请参考后面章节）

通过上述规定，在PA3 中所有的数据类型都统一为用 32 位整数实现，任何一个变量或者常量都占用 4 字节。

面向对象机制的实现[1]

面向对象机制的重要特点是封装、继承和多态三方面，这三方面在Decaf语言中分别体现为：类和对象的机制、类的继承机制、成员函数的动态分发机制。这三种机制除了需要通过语义分析的帮助来实现以外，还需要在运行时存储布局、函数调用等方面来加以实现。其中类和对象的机制和类的继承机制涉及到对象的内存表示方式，成员函数的动态分发机制涉及到成员函数的调用方法，在成员函数中访问成员域涉及到this关键字的实现方法，我们在下面对这三部分内容分别进行简单的介绍（同时请回忆C++中面向对象机制的实现方法）。

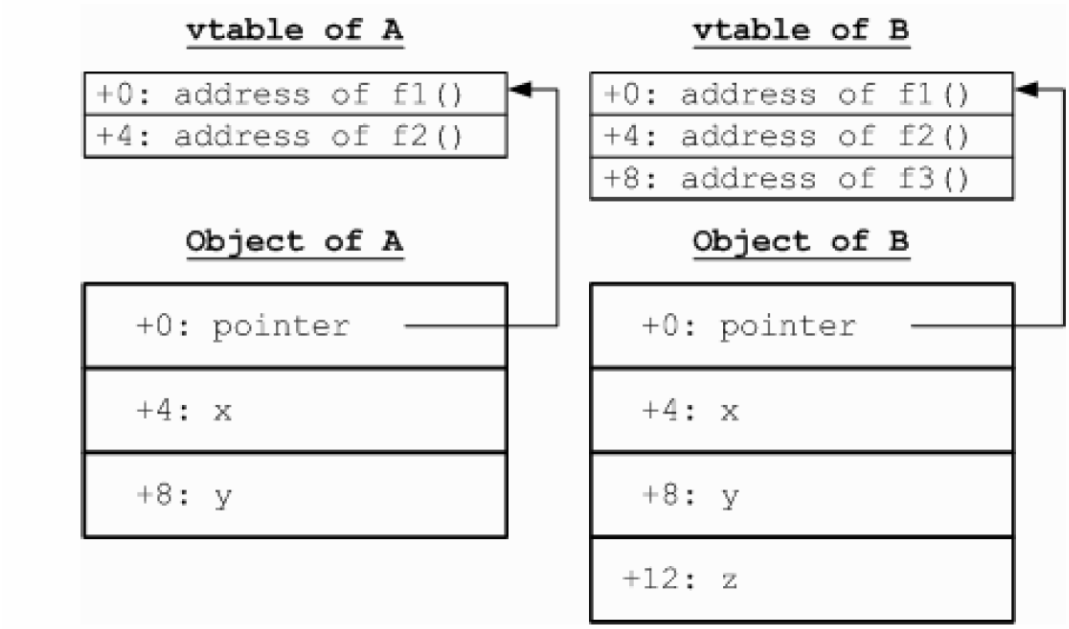
[1] 注：这一小节中有关虚表结构的是针对常规情形的一般性描述，本次实验的框架中 VTable 的结构与此不同，请参考课堂上第十讲的内容。

1、对象的内存表示及成员变量的访问方法

我们考虑如下类定义：

```
class A {
  int x;
  int y;
  int f1() {...}
  int f2() {...}
}
class B extends A {
  bool z;
  int f3() {...}
}
```

我们知道，f1、f2、f3 的地址都是存放在A或者B的虚函数表[2]中的，现在的问题是x、y、z是怎么存储的。由于不同的对象可以有不同的x、y和z的值，因此这些成员变量不能像成员函数那样存放在虚函数表中。一般来说这些域都存放在跟各对象相关联的内存块中，例如A的对象和B的对象的内存块的可能内容分别如图所示：



从图中可以看出，每一个对象实体都对应着一个记录这个对象状态的内存块，其中包括了这个对象的虚函数表指针和所有用于说明这个对象状态的成员变量。成员变量的排布顺序是：“辈分”越高的成员变量越靠前，例如从父类继承而来的成员变量总是在这个内存区域的前面，而子类特有的成员变量在这个内存区域的最后面，并且父类的成员变量的位置总是跟一个父类对象所对应的内存区域里面的情况一致的，这样做的目的是为了实现子类的对象能兼容于父类的对象（继承机制的一个表现）。

当访问一个对象的某个成员变量的时候，首先是通过这个对象的引用（在Decaf中引用通过指针实现，例如this）找到这块内存区域，然后再根据要访问的成员变量在这片内存区域中的偏移量对该成员变量进行访问的，由此可见，在Decaf中，访问一个对象的成员变量需要一次LOAD操作，而访问成员函数则由于需要通过首先按照访问成员变量的方式访问

[2] 注：这一部分所述的虚函数表（vtable）结构不同于 PA3 中的 vtable，后者是前者的扩充。

其虚函数表，然后在虚函数表中再次按照访问成员变量的方式拿到函数的入口指针，从而需要用两次LOAD操作。

2、 this关键字的处理

在Decaf语言中，成员函数的函数体内可以使用this关键字来获得对当前对象的引用。此外，在成员函数的函数体中对成员变量或者成员函数的访问实际上都隐含着对this的访问。例如，在writeName的函数体内使用了this关键字，则执行who.writeName()的时候this所引用的对象和变量who所引用的对象是相同的，同样道理，如果执行you.writeName()的话则writeName里面的this将引用you所指的对象。可见，在不同的上下文中，调用writeName的时候this所引用的对象是不同的，也就是说不可能在编译的时候确定下来，那么在一般的C++、Java等语言中是怎么实现这个功能的呢？

这里有一个技巧就是把who或者you作为writeName的一个实际参数在调用writeName的时候传进去，这样我们就可以把对this的引用全部转化为对这个参数的引用。例如，当声明Father.writeName的时候，我们在参数表的开头悄悄地加入一个叫做this的隐含参数：

```
int writeName(class Father this) {...}
```

并且在调用writeName时，在传实参的时候把消息接收方表达式（即you或者who）的值传进去，即等价于这样的函数调用：

```
who.writeName() -> call writeName(who);
```

这样，我们只要把writeName函数体内所有对this关键字的引用都转化为对this隐含参数的引用即可。这个过程中必须小心注意的是this隐含参数的位置跟调用函数的时候的传参顺序要相对应。

实际上，我们在PA2 中创建Function对象的时候已经自动完成了this参数的添加操作，在PA3 中我们只需要按照一般函数的调用方法来进行压参、求函数入口地址、调用即可(请注意this对应的实参值应该在是最后一个压进栈的参数，因为压参顺序与形参表顺序刚好相反)。

3、 虚函数表及成员方法的调用

虚函数表的目的是实现运行时函数地址绑定，即所谓的动态分发机制，例如以下Decaf代码：

```
class Father {
  int writeName() { print(1); ...}
  int smile() { print(2); ...}
}
class Son extends Father {
  int writeName() { print(3); }
  int laugh() { print(4); }
}
```

这里Father类定义了一个writeName()方法，而其子类Son使用新的writeName()覆盖了这个方法。然后考虑以下的代码片断：

```
class Father a;
class Son b;
class Father c;
a = new Father();
b = new Son();
c = b;
```

明显，执行a.writeName()的结果是输出 1，执行b.writeName()的结果是输出 3，但是执行c.writeName()的结果会是什么呢？虽然c被声明为Father的对象，但是实际上它是一个Son的对象，因此，按照Decaf语言规范，c.writeName()所调用的应当是Son的writeName()，即输出 3（跟C++的虚函数特点是一样的）。

这种行为在二进制层次上是怎么实现的呢？这里我们将采用一种叫做“虚函数表”的结构。我们为每一个类都创建一个存放成员函数入口地址的数组如下：

```
virtual table of Father:
+0: address of function writeName (the version prints 1)
+4: address of function smile
vitural table of Son:
+0: address of function writeName(the version prints 3)
+4: address of function smile
+8: address of function laugh
```

上图中每个虚函数表中都列出其对应的类的所有可访问的成员函数地址，成员函数地址前面的+0、+4 等等表示这个地址存放在虚表中的位置，例如+4 表示存放在离虚表开头 4 字节的地方。然后我们在a和b所指向的内存区域的开头分别放有指向Father和Son的虚表的指针，在每次通过a或者b调用writeName()的时候，我们都首先通过a或b开头的那个虚表指针找到对应的虚函数表，然后在表中找出偏移地址为+0 那一项对应的实际函数地址，调用之即可。

现在我们考虑c <- b的情况。由于Decaf的对象赋值采用引用赋值，因此这个赋值语句的效果仅仅是让c和b指向同一块内存区域。因此，按照上面的过程，当调用c.writeName()的时候，我们首先通过c所指向的内存区域找到对应的虚函数表（此时是Son的虚函数表），然后在这个虚函数表内找到writeName偏移量即+0 对应的那一项。我们发现这一项对应的函数地址是打印 3 的那个writeName()函数的地址，因此c.writeName()的调用结果是输出 3。

注意这里为了实现成员方法的继承，Son的虚函数表继承了Father的虚函数表中的smile那一项，并且为了保证子类兼容于父类，同名的函数在子类的虚表中的位置跟父类虚表中的位置是一样的，例如writeName和smile两个函数在Son虚表中处于+0 和+4 两个位置，这跟在Father的虚表中的情况一致；而原来Father中writeName的入口地址被Son版本的writeName的入口地址取代，以此实现成员函数的覆盖。

关于虚函数表的更多内容请参考：

http://en.wikipedia.org/wiki/Virtual_table

从上面关于虚函数表和成员方法的调用过程的介绍可以看出，Decaf中成员方法调用过程与一般C语言函数的调用过程不同之处仅在于通过虚函数表来查找函数入口地址以及加入了隐含的Decaf参数。

4、静态方法的调用

静态方法类似C语言的函数，直接得到地址调用，不查询虚表。

提示

这个阶段的内容比较乱，我们建议大家在实现的过程中先阅读一下给出的示例代码，然后自己先制定出一个实现顺序（可以依照测例一个个进行）。

请注意调试的时候应该自己使用一个非常简单的测试文件，对每种语句，分别实现和测试。PA3 初期切忌使用大而全的文件来进行调试，否则很容易陷入一片混乱当中。

PA3 的工作量和难度都比较大，但是只要早动手，阅读相关文档和代码对工作事先有所了解，抓紧时间，绝大多数同学都应该能够达到要求。

以下是助教给大家的一些建议：

1. 对于我们新增的五个语言特性，在pa3 实验中难度各有不同，其中case表达式和串行循环卫士语句更简单，建议大家从这两部分开始着手，能够更进一步熟悉pa3的过程。
2. 对于复数处理，如果我们给每个复数分配 64 位空间，那么需要找到文件中多处与内存分配相关的部分。另外，还需要注意在传递复数参数时的处理。