Decaf PA4 说明

任务描述

经过 PA1、PA2、PA3 的工作，我们已经实现了完整的 Decaf 编译器前端，事实上对于任何一种编程语言，其编译器前端均需要经历词法分析、语法分析、语义分析和中间代码生成这四个部分。获得 TAC 中间代码以后，我们便进入编译器的中端和后端部分，在这两部分中 TAC 中间代码将会被划分成一系列的基本块，并且根据基本块之间的跳转关系表示成控制流图。在控制流图层次上，编译器可能会对代码进行一系列的数据流分析（例如活跃变量分析、定值–引用分析等等）以及控制流分析（例如循环检测等），然后基于这些分析结果对代码进行一系列的优化（例如常量传播、公共表达式提取、循环代码外提、循环展开等等），然后依次经过指令选择、寄存器分配等步骤，最终转换成目标机器的汇编代码（或者机器代码）。

在 PA4 中，我们将实现 Decaf 编译器的数据流分析功能。数据流分析是现代编译器中绝大多数编译优化的基础。作为示例，实验框架中给出了活跃变量分析，即分析每一条 TAC 指令执行前后哪些变量是活跃的。所谓活跃变量，是指这个变量的当前内容在后面的代码中还会被使用，因此暂时不能把这个变量注销（即“杀死”一个变量）。在 Decaf 编译器中，活跃变量分析的结果将用于指导寄存器分配和临时空间的选择过程。

在 PA4 中，同学们自己需要完成的数据流分析功能是求解 DU（定值—引用）链或者 UD（引用—定值）链，二者选做其一。

假设在程序中某点 p 定义了变量 A 的值，从 p 存在一条到达 A 的某个引用点 s 的路径，且该路径上不存在 A 的其他定值点，则把所有此类引用点 s 的全体称为 A 在定值点 p 的**定值－引用链**（Definition-Use Chaining），简称 **DU 链**。课堂讲稿（2.4.2）中给出了一种关于 DU 链的求解方法，可供同学参考。

假设在程序中某点 u 引用了变量 A 的值，则把能到达 u 的 A 的所有定值点的全体，称为 A 在引用点 u 的**引用-定值链**（*Use-Definition Chaining*），简称 **UD 链**。课堂讲稿（2.4.1）中给出了一种关于 UD 链的求解方法，可供同学参考。

PA4 是 Decaf 编译器项目的最后一个阶段，完成这个阶段以后，结合我们提供的编译器后端部分代码（汇编指令选择、寄存器分配和栈帧管理），即可产生出适合实际的 MIPS 机器上的汇编代码，可以利用由美国 Wisconsin 大学所开发的 MIPS 模拟器 SPIM 来运行这些汇编代码。

下发给大家的实验框架会有两种输出，一个是数据流分析的结果（活跃变量分析），一个是最终的汇编代码。大家的任务是在数据流分析的结果中，除了能够输出活跃变量数据流信息外，进一步能够正确输出 DU 链或者 UD 链的信息。输出的格式可以自定（也可以参考活跃变量分析输出的格式），为方便评阅，要求在实验报告中举几个有代表性的例子，采用通俗的表现形式解释对你的编译器所测试生成的结果。

这阶段在测试时首先要求所有测试例子输出的汇编代码在 SPIM 模拟器上能够正常运行，其运行结果符合框架中原有的效果（即你的修改没有破坏原来的编译框架），然后再确保 DU 链或者 UD 链信息输出正确。PA4 和 PA3 的测试例子是一样的，最终运行结果也一样。

测试的方法与 PA1/2/3 类似，生成 decaf.jar 后，到 TestCases/S4 下面运行 runAll.py。对每一个 filename.decaf 文件，如果顺利，修改之前的编译器会在 output 下生成三个文件：

filename.dout：活跃变量数据流分析的结果

filename.s：MIPS 汇编文件，可用 SPIM 运行

filename.result：用 SPIM 运行 filename.s 的结果，除 SPIM 版本信息等无关紧要的内容外，它应与 result 下的相应文件相同。

其中 blackjack.decaf 仍然是“半隐藏”的样例，运行 runBlackjack.py 进行测试。

对于修改后的编译器，应当可以生成另外一个文件，记录 DU 链或者 UD 链信息的输出结果。**文件名分别是 filename.du 或 filename.ud**（选 DU 链是用前者，选 UD 链时用后者）。

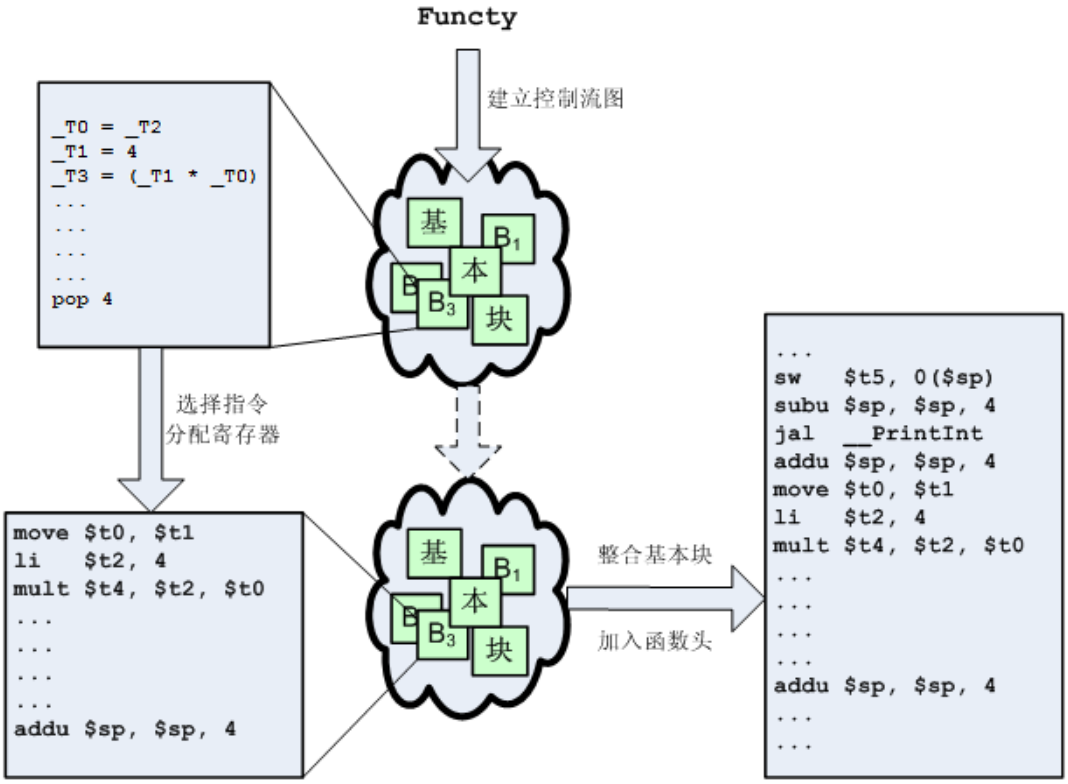
PA4 代码框架说明

文件/类	含义	说明
dataflow/BasicBlock	基本块定义	补充需要实现的部分
dataflow/FlowGraph	控制流图定义	补充需要实现的部分
machdesc/*	机器描述部分	不要修改
backend/*	编译器后端	**不要修改**
tac/*	TAC 语句	不需要修改
translate/*	从 AST 翻译到 TAC 的部分	PA3 工作，不需要修改
frontend/*	编译器最前端	PA1 工作，不需要修改
typecheck/*	语义检查部分	PA2 工作，不需要修改
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	不要修改
error/*	表示编译错误的类	不要修改
Driver	Decaf 编译器入口	需要增加所完成数据流分析（UD 链或 DU 链）的输出，**其他内容不要修改**
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

注：上面“不需要修改”的类是指不修改也可完成本次实验，但是你如果认为确实有必要，也可以适当修改；标明“不要修改”的类请不要动，尤其是红色标明的两处，不恰当的修改会影响得分。

在 Decaf 编译器中，最终的汇编代码生成是以函数为单位的，也就是 Functy。首先，我们会将一个 Functy 内部的 TAC 代码序列划分为一个个基本块，并建立控制流图，在控制流图上进行数据流分析。数据流分析完成后，Decaf 编译器会以整个 Functy 为单位进行寄存器分配，然后再针对每个基本块生成汇编代码，最后把这些汇编代码串起来，增加函数头和建立栈桢等操作，生成最终的 MIPS 汇编语言代码。下面是 Decaf 编译器中后端的

大大致工作原理图：



本阶段的测试例子会有两种输出，使用选项-l 3 可以输出数据流分析的结果（注意这个输出是无法在 tac 模拟器上的运行的），使用选项-l 4 可以输出 mips 汇编文件，这个文件可以通过 SPIM 模拟器运行和调试。

实际机器与汇编语言

对于大多数的程序员来说，他们编写程序的时候通常使用类似 C++这样的高级语言。使用这些语言编写程序的最大优势在于代码更侧重体现解题的逻辑，而不需要涉及物理机器的限制（例如操作数的类型、寄存器的数量、汇编指令格式等），这使得使用高级语言编写程序非常方便，但是高级语言所编写的程序不能直接在物理机器上运行。另一方面使用实际机器所能理解的语言进行编程的时候所受的限制是非常多的(正如通过逻辑门搭建电路的时候需要受到很多的限制)，我们可以先看看下面一段求 的程序（用 MIPS 机器指令表达）：

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00000110000010000000000000011101100
00100100100001000000001000001100000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111100000000000000001000
00000000000000000000100000100001
```

上面每一行的 32 位二进制串表示一个指令，整个程序一共 24 个指令，其中每个指令的某些位构成指令的操作码部分，即表示该指令进行什么操作；另一些位构成指令的操作数部分，即这些操作的对象是什么。对于大多数的程序员来说，直接编写这样的程序是非常痛苦的。为了让程序员能更方便地把思想编码为程序代码，人们使用助记符的方式来表达每一个指令，即汇编语言，其基本形式为：

助记符　操作数 0，操作数 1，操作数 2

该形式跟三地址码是非常相似的：助记符用于指示进行什么操作；操作数 0 通常是目标操作数，即用于存放操作结果；操作数 1 和操作数 2 是源操作数，即该操作对什么对象进行。例如：

```
add $r0, $r1, $r2
```

这条指令说的是把寄存器$r1 和$r2 的内容按照有符号数的运算方式相加，并把计算结果存到寄存器$r0 中。上面的例子用汇编语言表达如下图所示。

由于 PA4 中我们并不过多涉及汇编语言的编程知识，因此我们并不打算详细解释以上的内容。关于这部分内容，有兴趣的同学可以参考“Assemblers, Linkers and the SPIM Simulator”的有关章节。

从上面指令可以看出，汇编指令与 TAC 不同的是，TAC 中作为操作数的临时变量个数是无限的，因此在 PA3 中我们常常会遇到诸如_T235 这样的临时变量，但是在实际机器中，寄存器——特别是通用寄存器——的个数是有限的（在 MIPS 中 32 位通用寄存器有 18 个，在 Intel 公司的 80386 处理器中 32 位通用寄存器有 4 个，在 Sun 公司的 SPARC 处理器中通用寄存器可达两百多个），因此寄存器是一种十分宝贵的资源，不能像 TAC 那样随意使用。

```
        .text
        .align    2
        .globl    main
main:
        subu      $sp, $sp, 32
        sw        $ra, 20($sp)
        sd        $a0, 32($sp)
        sw        $0,  24($sp)
        sw        $0,  28($sp)
loop:
        lw        $t6, 28($sp)
        mul       $t7, $t6, $t6
        lw        $t8, 24($sp)
        addu      $t9, $t8, $t7
        sw        $t9, 24($sp)
        addu      $t0, $t6, 1
        sw        $t0, 28($sp)
        ble       $t0, 100, loop
        la        $a0, str
        lw        $a1, 24($sp)
        jal       printf
        move      $v0, $0
        lw        $ra, 20($sp)
        addu      $sp, $sp, 32
        jr        $ra


        .data
        .align    0
str:
        .asciiz   "The sum from 0 .. 100 is %d\n"
```

汇编指令与 TAC 不同的另一个地方是汇编指令实际上只是机器二进制码的一种助记形式，因此里面的操作数只有寄存器和立即数两种（立即数也可以表示内存地址），没有诸如虚函数表这样高级的数据对象，也没有林林总总的 int、long 等不同长度的操作数。除此以外汇编语言编程还有一系列的限制（例如立即操作数的大小、跳转距离等等）。

总之，用汇编语言表达算法还是相当不方便的，因此人们一般采用编译器来把用高级语言写的代码转换为上面的 0—1 机器代码或者是汇编代码，然后再在实际物理机器上运行。

在这个转换的过程中，编译器需要面对两个问题：一是选用什么样的机器指令来实现某种语句，二是如何把无数多个临时变量转换为有限个寄存器。第一个问题也称为指令选择，对于 MIPS 等 RISC 架构来说，指令选择比较简单，但是对于 x86 等 CISC 架构来说，由于可以选用多种指令来实现同一个功能，因此需要衡量不同指令之间的运行时间、空间和功耗等指标来进行选择。第二个问题一般叫做寄存器分配，即对于每个变量，选用什么寄存器来表示它。

对于操作顺序固定的程序而言，目前指令选择比较好的做法是基于树表示的动态规划算法（可参考 A.W.Appel 的“Modern Compiler Implementation in Java/C/ML”，即“虎书”），但是对于涉及到语句顺序调整的指令选择问题来说目前还没有很好的解决办法。

寄存器选择方面的内容我们将在后面的章节介绍。

在 PA4 中大家只需要了解什么是指令选择和寄存器分配，为什么编译器需要执行这两种处理等问题即可。

基本块与控制流图

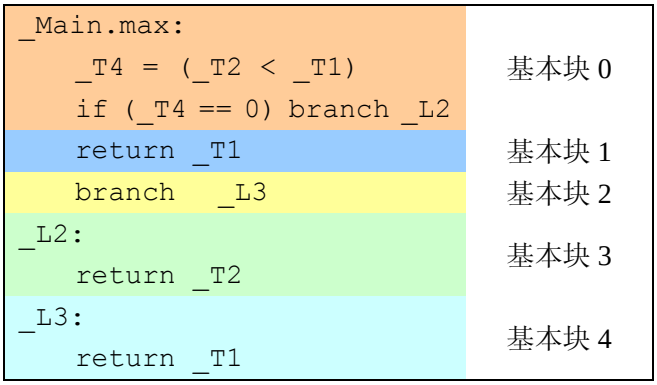
对于编译优化而言，PA2 中所进行的简单的语义分析是远远不足够的，为了进行更深入的优化，编译器需要了解程序语义的更多内容，例如一个变量的某个赋值在当前指令中是否有效、一个变量在当前指令以后是否还会被使用、当前运算指令的两个操作数是否都能够在编译的时候计算出来、循环体中某些代码是否能够提出到循环外面、循环次数是不是编译的时候已知的常数……

这些语义分析一般分为数据流分析和控制流分析两种。所谓数据流分析，是指分析某些变量的某个值满足什么性质，例如这个值能够影响到哪些指令、某个函数是否会被调用等；而所谓的控制流分析，是指分析程序的执行路径满足什么性质，例如是否包含多重循环、控制流图中是否包含强连通块等。

基本块（basic block）和控制流图（control-flow graph）是用于进行上述分析的数据结构。所谓的基本块，是指一段这样的代码序列：

1、除出口语句外基本块中不含任何的 Branch、Beqz、Bnez 或者 Return 等跳转语句（但可以包含 Call 语句）
2、除入口语句外基本块中不含任何的目标行号，即不能跳转到基本块中间
3、在满足前两条的前提下含有最多的连续语句，即头尾再纳入一条语句的话均会违反上面两条规则。

例如下面的代码中不同的基本块被使用不同的颜色标记出来：



也就是说，基本块内的代码执行过程总是从基本块入口开始，到基本块出口结束的，中间不会跳到别的地方或者从别的地方跳进来。

所谓控制流图是这样的一幅有向图：它以基本块作为结点，如果一个基本块执行完以后可能跳转到另一个基本块，则图中含有从第一个基本块指向后一个基本块的有向边。对于以 Branch 语句或者不以任何跳转语句结尾的基本块，其后继只有一个结点；对于以 Beqz 或者 Bnez 结尾的基本块，其后继含有两个结点，分别对应着跳转条件成立和不成立的时候的下一个基本块。不难想像，控制流图是一幅有向有环的图，其中的每一个环路都对应着程序中的一个循环结构。由于这幅图体现的是程序控制流的各种可能执行路径，因此这幅图称为控制流图。

对于优化编译器来说，建立控制流图是对代码进行优化的过程中必不可少的一步。已知一个操作序列，如何根据这个操作序列建立对应的控制流图呢？我们通常是划分为两步进行：第一步划分基本块，第二步建立基本块之间的连接关系。

基本块的划分算法非常简单：我们从头到尾扫描操作序列；当遇到一个行号标记而且存在跳转语句跳转到这个行号的话，则新起一个基本块（并且结束前一个基本块）；当遇到 Branch、Beqz、Bnez 或者 Return 等跳转语句的时候我们结束当前基本块，然后新起一

个基本块。如果一个基本块不以跳转语句结尾，则我们在它最后加入一个 Branch 语句，跳转到下一个基本块。当整个操作序列扫描完毕以后所有基本块都应该已经被标识出来了。

在划分好基本块以后，我们需要建立控制流图：从头到尾依次扫描所有的基本块，如果当前基本块以 Branch 结尾，则我们在当前基本块和目标基本块之间加入一条边；如果当前基本块以 Beqz 或者 Bnez 结尾，则我们在当前基本块和跳转条件成立的时候的目标基本块之间加入一条边，并且标记这条边为 0，然后在当前基本块和跳转条件不成立的时候的目标基本块之间加入一条边，并且标记这条边为 1；如果当前基本块以 Return 结尾，则不用加入新的边。当所有的基本块都扫描完毕以后控制流图即可建立完毕。建立控制流图是进行控制流分析的基本操作。

在 PA4 中，我们总是规定编号为 0 的基本块是整个函数的入口。

活跃变量和活跃变量方程

从编译器前端出来的中间代码中，我们对所允许使用的临时变量的个数并没有做任何限制，但是在实际机器中，寄存器的数量是有限的，因此我们需要想办法把这些无限多的临时变量“塞”到有限个寄存器里面：如果两个临时变量不会在同一条指令中被用到，那么我们可以让这两个临时变量使用同一个寄存器。根据这样的原则，大多数的临时变量都可以用有限的几个寄存器对应起来，而“塞不下”的那些临时变量，则可以暂时保存到内存里面（请注意汇编语言中不能两个内存地址同时运算，而且访问内存的时间代价比访问寄存器的时间代价要大得多，因此临时变量应该尽可能都放在寄存器中）。

由于一个寄存器在确定的时刻只能容纳一个变量，因此为了把若干个变量塞到同一个寄存器里面，我们需要知道各个变量分别在什么时候可以被扔出寄存器外面（以便腾出寄存器给别的变量）。此时我们需要用到活性分析（liveness analysis），或者称为“活跃变量分析”。

一个变量在某个执行点是活的（也叫“活跃”、live），是指该变量在该执行点处具有的变量值会在这个执行点以后被用到，换句话说，就是在该执行点到给这个变量重新赋值的执行点之间存在着使用到这个变量的语句。活性分析是指分析每一个变量在程序的每一个执行点处的活跃情况，通常是通过计算出每个执行点处的活跃变量集合来完成。

下面代码中每行语句右边都给出了执行完该语句的时候的活跃变量集合：

TAC 代码	活跃变量集合
_T6 = 4	{_T6}
parm _T6	{}
_T7 = call _Alloc	{_T7}
_T9 <- VTBL <Main>	{_T7, _T9}
*(_T7 + 0) = _T9	{_T7, _T9}
parm _T7	{_T9}
_T10 <- *(_T9 + 4)	{_T10}
_T11 <- call _T10	{_T11}
return _T11	{}

例如第 5 行代码给_T9 赋值以后，变量_T9 一直都是活的直到第 8 行，即最后一条使用_T9 的命令以后为止。

一般来说，活性分析是通过求解活跃变量方程来完成的。为了介绍活跃变量方程的概念，我们需要先引入下面四种针对基本块的集合：

1、Def 集合：一个基本块的 Def 集合是在这个基本块内被定值的所有变量。所谓的定值（definition），就是指给变量赋值，例如一般的赋值语句给左边变量定值，又例如加法操作语句给目标变量定值等等（但是 Store 语句并不给任何变量定值）。

2、LiveUse 集合：一个基本块的 LiveUse 集合是在这个基本块中所有在定值前就被引用过的变量，包括了在这个基本块中被引用到但是没有被定值的那些变量。

3、LiveIn 集合：在进入基本块入口之前必须是活跃的那些变量。

4、LiveOut 集合：在离开基本块出口的时候是活跃的那些变量。

为了直观地理解上面四个集合，我们看看下面的例子：

```
parm   _T6
call _PrintInt
_T28 = 1
return _T28
```

在上面基本块中_T28 被定值了，而_T6 在使用的时候并没有被定值，因此：

Def = {_T28}

LiveUse = {_T6}

如果离开这个基本块以后所有的变量都不再活跃，即 LiveOut = {}，那么在进入这个基本块之前至少_T6 必须是活跃的（因为要用到它的初值），故 LiveIn = {_T6}。

有了基本块的这四个集合的概念，我们可以控制流图中每个基本块 都应当满足的活跃变量方程：

$$\begin{cases} LiveOut(B) = \bigcup_{B_i \in succ(B)} LiveIn(B_i) \\ LiveIn(B) = LiveUse(B) \cup (LiveOut(B) - Def(B)) \end{cases}$$

该方程说的是一个基本块 的 LiveOut 集合是其所有后继基本块的 LiveIn 集合的并集，而且 LiveIn 集合是 LiveUse 集合的变量加上 LiveOut 集合中去掉 Def 集合以后的部分。

这个方程的直观意义是：

1、在一个基本块的任何一个后继基本块入口处活跃的变量在这个基本块的出口必须也是活跃的

2、在一个基本块入口处需要活跃的变量是在该基本块中没有定值就被使用的变量，以及在基本块出口处活跃但是基本块中没有定值过的变量（因为它们的初值必定是在进入基本块之前就要具有的了）

解这个方程有很多算法，下面是一种比较简单的方法：

```
for i <- 1 to N do compute Def[Bi] and LiveUse[Bi];

for i <- 1 to N do LiveIn[Bi] <- φ;
changed <- true;
while (changed) do {
  changed <- false;
  for i <- N downto 1 do {
    LiveOut[Bi] <- ∪LiveIn[s]; where s ∈ succ(Bi)
    NewLiveIn <- LiveUse[Bi] ∪ (LiveOut[Bi] - Def[Bi]);
```

```
    if (LiveIn[B_i] ≠ NewLiveIn) then {
      changed <- true;
      LiveIn[B_i] <- NewLiveIn;
    }
  }
}
```

上面算法中 succ(B_i)表示 在控制流图中的所有后继基本块。上面的算法是一种迭代算法，实际上大部分的数据流方程（数据流分析问题通常都是解数据流方程的问题）都可以采用这种迭代算法来计算。为简单起见，我们在这里并不证明这个算法的收敛性。

上面算法中并没有给出计算 Def 集和 LiveUse 集合的方法，然而根据前面关于这两个集合的定义，相信同学们不难想出高效的计算方法。

获得了每个基本块的 LiveIn 和 LiveOut 集合以后，我们需要进一步地计算每个 TAC 语句的 LiveIn 和 LiveOut 集合。如果我们把基本块的所有 TAC 语句分别看成是一个独立的基本块，则不难想像，前面提到的活跃变量方程仍然有效，不同的是这样的“控制流图”有以下三种特点：

1、每个节点的出度都是 1，也就是说 $LiveOut(B) = LiveIn(succ(B))$

2、由于每个结点只含有一个语句，因此其 Def 集要么是空集，要么只含有一个元素

3、由于每个节点中第一个语句里面所引用到的所有变量在使用的时候都未经定值，因此其 LiveUse 集合就是源操作数中所有的变量。

基于上面三个特点，已经求出基本块的 LiveOut 集合的前提下我们很容易设计出计算基本块内所有 TAC 的 LiveOut 集合的算法（请大家自行设计）。另外特点（1）表明了对于每个 TAC 语句，实际上只需要记录其 LiveOut 集合即可，因此在 PA4 代码框架中每个 TAC 结点我们只保存了 LiveOut 集合。

当基本块和 TAC 的数据流分析都完整以后，PA4 部分的工作即告完成。在 PA4 的输出信息中，每个基本块的 Def、LiveUse、LiveIn、LiveOut 集合均会被输出，而且每条 TAC 语句的 LiveOut 集合将会在它的右边打印出来。

在完成 PA4 的过程中，大家可能需要事先总结出出现在基本块内的各种 TAC 语句中哪些操作数是作为源操作数使用的，哪些操作数是作为目标操作数来保存计算结果的。在整理这个总结的过程中需要注意：

1、基本块内不会出现 Mark、Memo、Jump、JZero 和 Return 语句

2、对于 DirectCall 和 IndirectCall 语句，其目的操作数有可能为 NULL

3、事先搞清楚每种 tac 语句中哪些是源操作数，哪些是目的操作数

寄存器分配和栈帧管理

理想的寄存器分配结果应当是所有的变量都有合适的寄存器跟它相对应，而且在程序执行过程中的任何时刻，不会出现寄存器分配冲突（即两个同时有效的变量分配到同一个寄存器，特别是同一条指令中用到的两个变量分配到同一个寄存器）。事实上，理想寄存器分配问题是 NP 完备的（可归约成 3-SAT 问题），这意味着对于一个含有几百个变量的程序，为了获得理想的寄存器分配方案，编译器需要消耗大量的时间来进行运算。因此，实际的编译器中均只追求获得该问题的近似最优解。

一般编译器中的寄存器分配方法可分为两类，一类是全局寄存器分配，另一类是局部寄存器分配。全局寄存器分配是指分配寄存器的时候把控制流图中所有的基本块一起考虑，局部寄存器分配是对每个基本块分配进行寄存器分配。

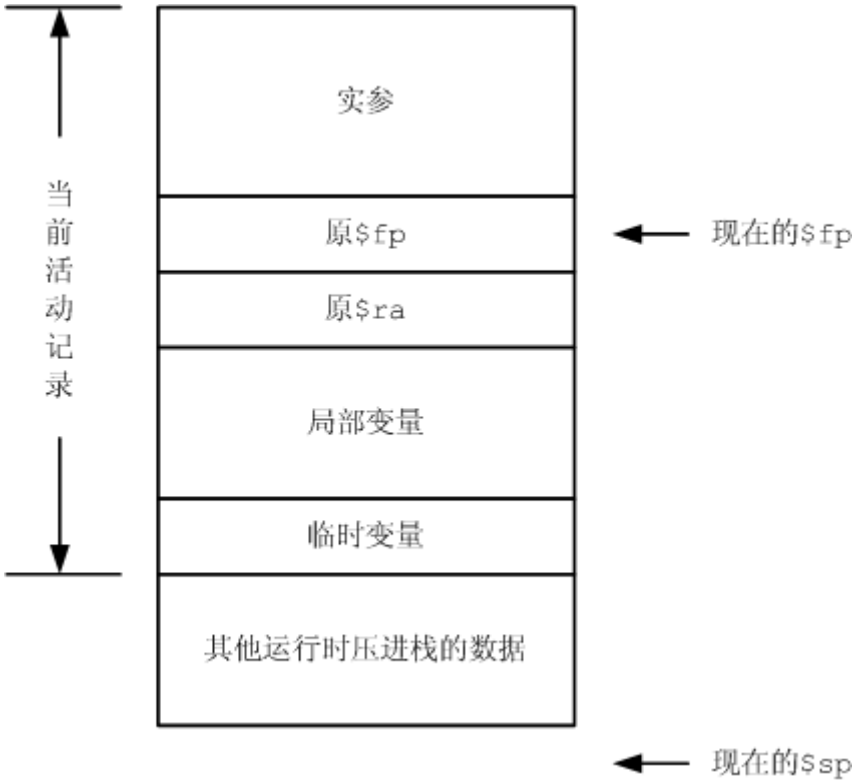
一般编译器中的寄存器分配方法可分为两类，一类是全局寄存器分配，另一类是局部寄存器分配。全局寄存器分配是指分配寄存器的时候把控制流图中所有的基本块一起考虑，局部寄存器分配是对每个基本块分配进行寄存器分配。

全局寄存器分配通常采用图着色算法，其基本思想是以一个变量作为图的一个结点，如果两个变量需要同时参与同一条指令的运算的话，则这两个变量对应的结点之间连一条边。这样我们便获得所谓的干涉图（interference graph）。然后全局寄存器分配的问题便转化为“如何采用 K 种颜色对干涉图的结点着色，使得相邻的结点总着以不同颜色”的问题，其中 K 是实际机器中的通用寄存器数量，可以采用例如模拟退火、遗传算法等优化方法来解决。但是实际使用中，为了图着色算法更加高效起见，通常不使用单一的变量作为干涉图的一个结点，而是用一系列相交的 DU 链（定值—引用链）构成的同色网（web）作为一个结点来构造干涉图。另外，只采用 K 种颜色来染色通常是不可能的，此时往往是先求一种染色方案，使得所用的颜色数最少，然后选择保存到内存的代价最小的那些变量来保存到内存（从而该结点可以着以一种特殊的颜色）。

Decaf 采用一种很简单的局部寄存器分配策略，在基本块开头，把 liveIn 的变量 load 进寄存器，基本块内为变量分配寄存器，在离开基本块时，将 liveOut 的变量 store 出去。

局部寄存器分配是对每个基本块分别进行寄存器分配的方法。局部寄存器分配的实现有多种，在 Decaf 编译器中采用了一种基于活性分析结果的做法，其基本思想是：从头到尾扫描基本块中的指令；对于每条指令的源操作数，我们查看是否已经放在寄存器里面，如果不是则分配一个寄存器并从栈帧中把该变量加载进寄存器中；对于每条指令的目标操作数，如果还没有跟某个寄存器关联，则我们分配一个新的寄存器给它。当分配寄存器的时候，我们首先看看有没有寄存器尚未跟某个变量关联，有则选择该寄存器作为分配结果，否则看看有没有寄存器所关联的变量在当前已经不是活跃变量了，有则选择该寄存器作为分配结果，否则说明当前所有寄存器所关联的变量都是活跃的，则需要把某个寄存器所关联的现在暂时不用的变量“扔”到栈帧（内存的一部分）中，从而腾出这个寄存器。

这里所说的栈帧（stack frame）是我们在 PA3 文档中提到的运行时存储布局的相关内容。栈帧是分配在栈上的内存区域，如图所示：



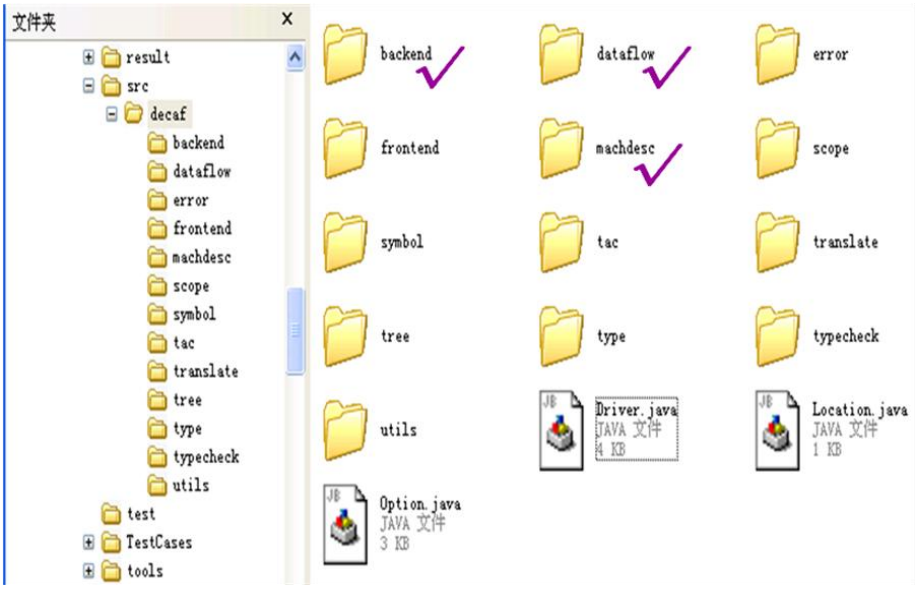
图中寄存器$fp 和$sp 所指位置之间的区域即当前函数的栈帧。在 Decaf 编译器中，由于局部变量和临时变量在使用上实际上并没有区别，因此我们把这两个区域混合使用。在 Decaf 编译器中，当发现寄存器不够用时，就会使用栈帧来存放临时变量和局部变量。其具体实现相对复杂，这里就不再叙述了。

栈帧是函数运行所需要的上下文的一部分，在进入函数的时候需要建立自己的栈帧，在退出函数的时候需要销毁自己所建立的栈帧。由于栈帧的初始大小在编译的时候是能够确定下来的，因此建立栈帧的时候只需要把旧的栈帧首地址（存放在$fp 中）保存起来，然后用 $fp 保存旧栈帧的尾地址（存放在$sp 中），适当修改$sp 寄存器的内容让它指向新栈帧的尾地址即可。可见，每个栈帧在栈上是头尾相接的，它们把栈区分为一个一个的窗口（frame），这正是“栈帧”（stack-frame）这个名字的来源。

在 Decaf 中，由于栈桢的大小在编译期的时候是可以计算出来的，因此 fp 实际上是不需要的，因为移动和恢复 sp 的时候，可以简单的通过加减立即数来做到，并不需要事先保存。在实际的编译器里，在栈大小小于 65536 个字节，并且没有 alloca 之类会动态分配栈空间的操作时，也是不会使用 fp 的。

注：以上若干方面有助于对实验框架（特别是中后端）的理解，但许多都是课堂上或其他课程中大家已经了解的基本内容，而且与本阶段实验也没有必然联系，因此仅供选择性阅读。以下列举了实验框架中与本阶段紧密相关的基本块与流图部分的代码结构，供同学参考。

源码的中后端目录结构



基本块

基本块类 BasicBlock（见 decaf.dataflow.BasicBlock）的定义框架为：

```
public class BasicBlock {
    public int bbNum;                // 基本块的编号
    public enum EndKind {
        BY_BRANCH, BY_BEQZ, BY_BNEZ, BY_RETURN
    }
    public EndKind endKind;          // 出口类别
    public int inDegree;             // 从入口基本块开始计数的第 inDegree 个基本块
    public Tac tacList;              // 基本块的 Tac 语句序列
    public Label label;              // 当前基本块汇编码起始位置的标号
    public Temp var;                 // 出口语句 RETURN, BEQZ, BNEZ 的操作数变量
    public Register varReg;          // 出口语句 RETURN, BEQZ, BNEZ 的操作数寄存器
    public int[] next;               // 后继基本块（最多 2 个）的编号
    public boolean cancelled;        // 标记为死代码的基本块
    public boolean mark;             // 标记当前基本块的汇编码是否已输出
    public Set<Temp> def;            // 基本块的 Def 集合
    public Set<Temp> liveUse;        // 基本块的 LiveUse 集合
    public Set<Temp> liveIn;         // 基本块的 LiveIn 集合
    public Set<Temp> liveOut;        // 基本块的 LiveOut 集合
    public Set<Temp> saves;          // 离开基本块时须保存的寄存器集合
    private List<Asm> asms;          // 基本块的汇编指令序列
    ……
    public void computeDefAndLiveUse() {     // 计算 Def 和 LiveUse 集合
        ……
```

```
        }
        ……
        public void analyzeLiveness() {        // 基本块内的活跃变量分析
            ……
        }
        ……
    }
```

基本块编号 bbNum 会在标记基本块（调用流图类 FlowGraph 的 markBasicBlocks 函数）时设置。

每个基本块的出口类别 endKind，Tac 语句序列 tacList，出口语句 RETURN、BEQZ 和 BNEZ 的操作数变量 var 以及后继基本块（最多 2 个）的编号 next 会在由基本块生成流图的过程（调用类流图类 FlowGraph 的 gatherBasicBlocks 函数）时逐一定义。

Def 集合 def 以及 LiveUse 集合 liveUse 是基本块的固有属性，由这个类（BasicBlock）中的函数 computeDefAndLiveUse 计算并设置。在 computeDefAndLiveUse 得到 liveUse 时，同时将其置为基本块 LiveIn 集合 liveIn 的初值。

在流图中，每个基本块的 LiveOut 集合 liveOut 和 LiveIn 集合 liveIn，是由流图范围内的活跃变量数据流分析得出的，即调用流图类 FlowGraph 的 analyzeLiveness 函数。

这里值得区别一下流图范围内的活跃变量数据流分析与基本块内的活跃变量数据流分析，其函数名都是 analyzeLiveness，但属于不同的类。前者属于类 FlowGraph，用于计算基本块为单位的 LiveOut 集合和 LiveIn 集合。后者属于类 BasicBlock，用于计算 Tac 语句为单位的 LiveOut 集合和 LiveIn 集合(实际上,每个 Tac 语句只记录了 LiveOut 的值,其 LiveIn 的值可由 Tac 语句的构成及其 LiveOut 的值计算得到)。

inDegree 表示当前基本块是从入口基本块开始计数的第 inDegree 个基本块。若 cancelled 为 true，则表明当前基本块已被标记为死代码的基本块，即从入口基本块不可达。inDegree 和 cancelled 在流图类 FlowGraph 的 simplify 函数中使用，该函数通过删除不可达的基本块对流图进行化简。

label 将被设置为当前基本块汇编码起始位置的标号，mark 标记当前基本块的汇编码是否已输出。参见 decaf.backend.Mips。

varReg 表示分配给出口语句 RETURN、BEQZ 和 BNEZ 操作数（如果有的话）的寄存器，在寄存器分配时设置，在生成函数体代码时使用。

saves 表示离开基本块时须保存的寄存器中的变量集合，即基本块出口处已分配寄存器的活跃变量的集合，在寄存器分配时设置，在生成代码时使用。

私有属性 asms 被本类构造函数中被初始化为

```
asms = new ArrayList<Asm>();
```

在后续代码生成阶段，分别通过函数 getAsms 和 appendAsm 来访问和修改（添加一条指令 asm）基本块中的 asms。

流图

流图类 FlowGraph（见 decaf.dataflow.FlowGraph）的定义框架为：

```
public class FlowGraph implements Iterable<BasicBlock> {
    private Functy functy;              // 流图对应的函数
    private List<BasicBlock> bbs;       // 流图包含的基本块集合
    public FlowGraph(Functy func) {
        this.functy = func;
        deleteMemo(func);               // 去掉 Memo 记录
        bbs = new ArrayList<BasicBlock>();
        markBasicBlocks(func.head);     // 标记基本块
        gatherBasicBlocks(func.head);   // 生成流图
        simplify();                     // 简化流图（死代码删除）
        analyzeLiveness();              // 基本块为单位的活跃变量分析
        for (BasicBlock bb : bbs)  {
            bb.analyzeLiveness();       // 基本块内语句为单位的活跃变量分析
        }
    }
    private void deleteMemo(Functy func) {
        ……
    }
    private void markBasicBlocks(Tac t) {
        ……
    }
    private void gatherBasicBlocks(Tac start) {
        ……
    }
    public void analyzeLiveness() {
        ……
    }
    public void simplify() {
        ……
    }
    ……
}
```

每个函数对应一个流图。流图类构造函数描述的主要流程是：先从函数入口开始标记出它所包含的所有基本块（调用 markBasicBlocks），然后再从入口基本块（对应函数入口语句的基本块）开始逐步建立对应的流图（调用 gatherBasicBlocks）。其次，还辅以下列操作：

函数 simplify 删除从入口基本块不可达的那些基本块，以此对流图作初步的化简。

函数 analyzeLiveness 实现以基本块为单位的活跃变量分析，计算出流图中每个基本块的 LiveOut 集合和 LiveIn 集合。

构造函数最后的循环，用于计算每个基本块内部以 Tac 语句为单位的 LiveOut 集合（通过调用类 BasicBlock 的 analyzeLiveness 函数）。

此外，函数的 Memo 记录不参与代码生成（而是为 TAC 模拟专用），因而标记基本块建立流图之前先调用 deleteMemo 将其剔出。

提示

这个阶段要涉及的概念比较多，而且在一定程度上超前于课堂内容，因此大家需要较多的时间来学习这些内容，同时建议大家参考课程教案或课程参考书的相关内容。

虽然在本编程作业中不要求对 MIPS 体系架构的深入了解，但是有兴趣的同学仍可以参考 J.R.Larus（SPIM 模拟器的作者）的“Assemblers, Linkers, and the SPIM Simulator”的有关章节。更进一步的内容可参考 D.Sweetman 的“See MIPS Run”（需要有体系结构的基本知识，需要对 MIPS 结构有一定的了解）。

本阶段代码量不算多，但需要了解代码框架中相应的部分，并进行一定程度的设计，因此仍然需要花费不少时间和精力，大家选做时应考虑自己的实际情况。

本阶段的输出数据很多，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，待基本实现完毕以后再使用更加复杂的文件，否则将会陷入大量输出数据中难以理清思路。