# 编译原理期末整理

## 静态链和逃逸分析

```
/* A program to solve the 1-N queens problem */
let
        var maxN := 10
        var emptySymbol := " ."

        function printSquare(hasQueen:int) =
                let
                        var queenSymbol := " 0"
                in print(if hasQueen then queenSymbol else emptySymbol) end

        function nQueens(N:int) =
                let
                        type intArray = array of int

      var row := intArray [ N ] of 0
                        var col := intArray [ N ] of 0
      var diag1 := intArray [N+N-1] of 0
      var diag2 := intArray [N+N-1] of 0

      function printBoard() =
         (for i := 0 to N-1
                        do(for j := 0 to N-1
                                do printSquare(col[i]=j); print("\n"));
         print("\n"))

      function try(c:int) =(
        if c = N then printBoard()
        else for r := 0 to N-1
          do if row[r]= 0 & diag1[r+c]=0 & diag2[r-c+N-1]=0
                then (row[r]:=1; diag1[r+c]:=1; diag2[r-c+N-1]:=1;
              col[c]:=r;
              try(c+1);
              row[r]:=0; diag1[r+c]:=0; diag2[r-c+N-1]:=0)
      )
        in try(0) end
in
        for n := 1 to maxN
                do(print("Solving ");printi(n);print(" queens problem...\n\n");
       nQueens(n);
       print("\n"))
end
```

> 1. Suppose that we implement the nested functions using an access link. Please draw the access link in each of the activation records on the stack based on the following function

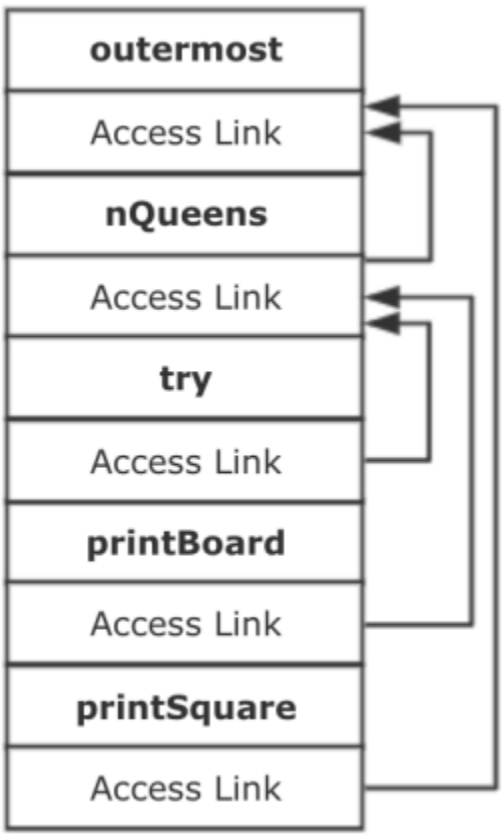
> invocations. (9')
> a. nQueens->try->try
> b. nQueens->try->printBoard
> c. nQueens->try->printBoard->printSquare

**画出函数层次分级图**

- outermost
  - printSquare
  - nQueens
    - printBoard
    - try

**根据调用函数顺序画出栈结构**
**每层的 *Access Link* 指向上一层次函数的 *Access Link***

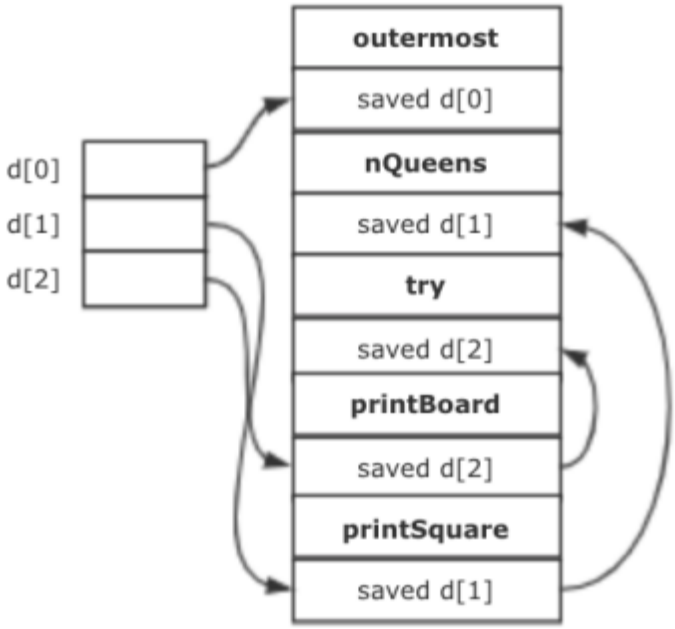
e.g.




> 2. Suppose that we implement nested functions using displays. Please draw the display in each of the activation records on the stack based on the following function invocations. (9')
> a. nQueens->try->try
> b. nQueens->try->printBoard
> c. nQueens->try->printBoard->printSquare

**根据第一问中的 *Access Link* 指向，来确定 d 的标号和指向**
**outermost 的 *Access Link* 总是 d[0]**
**其他函数看箭头所指向的上一层 *Access Link* 来确定 d 的标号**
**左侧 d 的直接指向总是所对应的栈的最深的位置处**

e.g.




> 3. Determine whether the following local variables can escape and explain your reasons.(12')
> | Variable | Escape(Y | N) | Your Reason |
> | :--: | :--: | :--: |
> | maxN (Line 4) | N | 没有被嵌套的函数访问，而且是简单的数据类型，能放入寄存器中 |
> | emptySymbol (Line 5) | Y | 字符串变量是指向字符串的指针，虽然可以存放寄存器中，被嵌套的 printSquare 函数访问 |
> | queenSymbol (Line 9) | N | 字符串变量是指向字符串的指针，且只被当前函数所访问 |
> | col (line 17) | Y | 被嵌套的 try 函数访问 |

**如果一个变量是传地址实参，或者它被取了地址（使用 C 语言中的 & 操作），或者内次的嵌套函数对其进行了访问，我们则称该变量是逃逸的（escape）**
**tiger 中传地址实参指的是函数参数中有 *Record* 或者 *Array* 类型**
**tiger 中没有取地址操作**
**tiger 中嵌套是最常见造成变量逃逸的原因**

# 活跃分析和寄存器分配

```
int nfactor (int n) {
  if (n <= 1) return 1;
  return n * nfactor(n - 1);
}
```

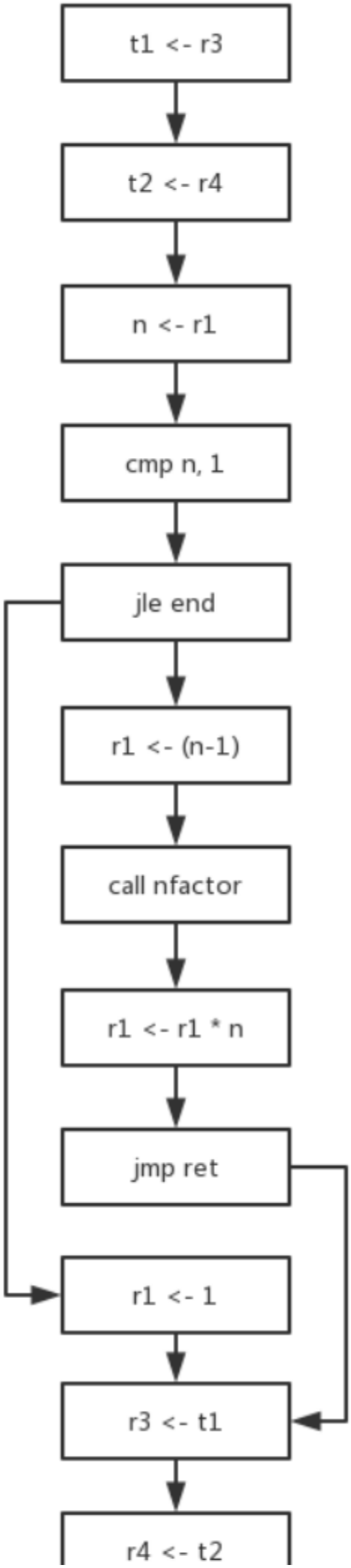
```
nfactor:
    t1 <- r3
    t2 <- r4
    n <- r1
    cmp n, 1
    jle end
    r1 <- (n - 1)
    call nfactor
    r1 <- r1 * n
    jmp ret
        end:
                r1 <- 1
        ret:
                r3 <- t1
                r4 <- t2
                return
```

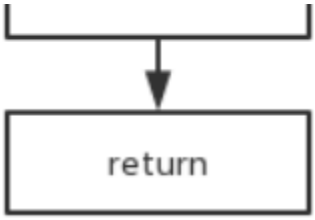
> Several things are worth noting in the instructions:

- There are **four** hardware registers in all.
- The compiler passes parameters using registers. (The first parameter goes to **r1**)
- **r1**,**r2** is **caller-saved** while **r3**,**r4** are **callee-saved**.
- The **return value** will be stored in **r1**.
- **t1**, **t2**, **n** are temporary registers.

> 1. Draw a control flow graph instruction-by-instruction. (4')

**有 jmp 则需更改那根线的指向，有 je、jne、jl、jg、jle、jge 则需多加一根线指向可能跳转的位置的下一条指令，其他的按照指令出现顺序画即可**


t1 <- r3
t2 <- r4
n <- r1
cmp n, 1
jle end
r1 <- (n-1)
call nfactor
r1 <- r1 * n
jmp ret
r1 <- 1
r3 <- t1
r4 <- t2




> 2. Fill up the following def/use/in/out chart. (10')

**def：对变量或临时变量的赋值。**
**use：出现在赋值号右边（或其他表达式中）的变量。**
**活跃的：存在一条边从这条边通向该变量的一个 use 的有向路径，并且此路径不经过该变量的任何 def。**
**in：如果一个变量在一个结点的所有入边上均是活跃的。**
**out：如果一个变量在一个结点的所有出边上均是活跃的，则该变量在该结点是出口活跃的。**
**pred[n]：结点 n 的所有前驱结点的集合。**
**succ[n]：结点 n 的所有后继结点的集合。**

活跃分析的数据流方程

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{3}$$

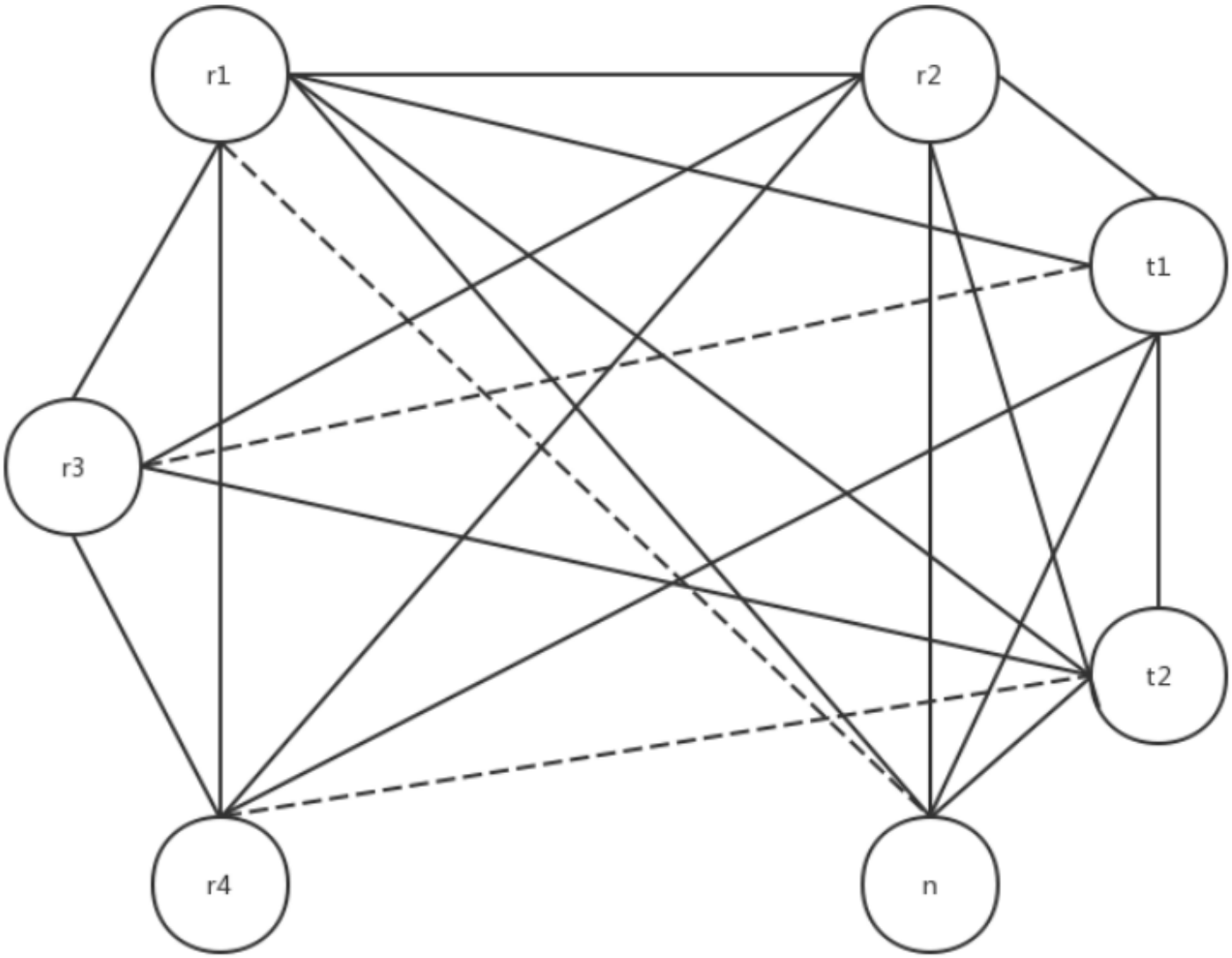
$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{4}$$

| Instr | def | use | in | out |
|---|---|---|---|---|
| t1 <- r3 | t1 | r3 | r1, r3, r4 | t1, r1, r4 |
| t2 <- r4 | t2 | r4 | t1, r1, r4 | t1, t2, r1 |
| n <- r1 | n | r1 | t1, t2, r1 | t1, t2, n |
| cmp n,1 | | n | t1, t2, n | t1,1 t2, n |
| jle end | | | t1, t2, n | t1, t2, n |
| r1 <- (n – 1) | r1 | n | t1, t2, n | r1, t1, t2, n |
| call nfactor | r1, r2 | r1 | r1, t1, t2, n | r1, t1, t2, n |
| r1 <- r1 * n | r1 | r1, n | r1, t1, t2, n | r1, t1, t2 |
| jmp ret | | | r1, t1,t2 | r1, t1, t2 |
| r1 <- 1 | r1 | | t1, t2 | r1, t1, t2 |
| r3 <- t1 | r3 | t1 | r1, t1, t2 | r1, r3, t2 |
| r4 <- t2 | r4 | t2 | r1, r3, t2 | r1, r3, r4 |
| return | | r1, r3, r4 | r1, r3, r4 | r1, r3, r4 |

- caller-saved 需要放在 call 指令的 def 处，传递函数参数的寄存器需要放在 call 指令的 use 处。
- return 指令的 def 和 use 一定为空，use in out 一定相同且为 callee-saved 、存放函数返回值的寄存器。

> 3. Draw the interference graph for the program. Please use dashed lines for move edges and solid line for real interference edges. (10')

- **对于任何对变量 $a$ 定值（def 集合）的非传送指令，以及在该指令处是 出口活跃（out 集合） 的变量 $b_1, \ldots, b_j$，添加冲突边 $(a, b_1), \ldots, (a, b_j)$。**
- **对于传送指令 $a \leftarrow c$，如果变量 $b_1, \ldots, b_j$ 在该指令处是 出口活跃（out 集合） 的，则对每一个不同于 $c$ 的 $b_i$ 添冲突边 $(a, b_1), \ldots, (a, b_j)$。**
- **机械寄存器之间互相之有一定冲突边。**




> 4. Adopt the graph-coloring algorithm to allocate registers for temporaries. (15')

流程：

1. 构造：构造冲突图，并用虚线指出传送有关的结点（即传送指令的源操作数和目的操作数之间连接虚线）。
2. 简化：每次一个地从图中删除 **低度数（度数 $<$ K）**、**传送无关** 的结点。
3. 合并：对一条传送指令的源操作数和目的操作数进行合并成为新的结点，并且遵守以下任一原则即可。
    - Briggs：结点 $a$ 和 $b$ 合并产生的结点 $ab$ 的 **高度数（度 $\geq K$）**邻结点的个数少于 $K$。

- George：对于 $a$ 的任一邻居 $t$，要么 $b$ 与 $t$ 已经有冲突，要么 $t$ 是 **低度数（度 $<$ $K$ 的结点。**

**重复进行 *简化* 和 *合并* 过程，直到仅剩下 *高度数结点* 或 *传送有关结点* 为止。**

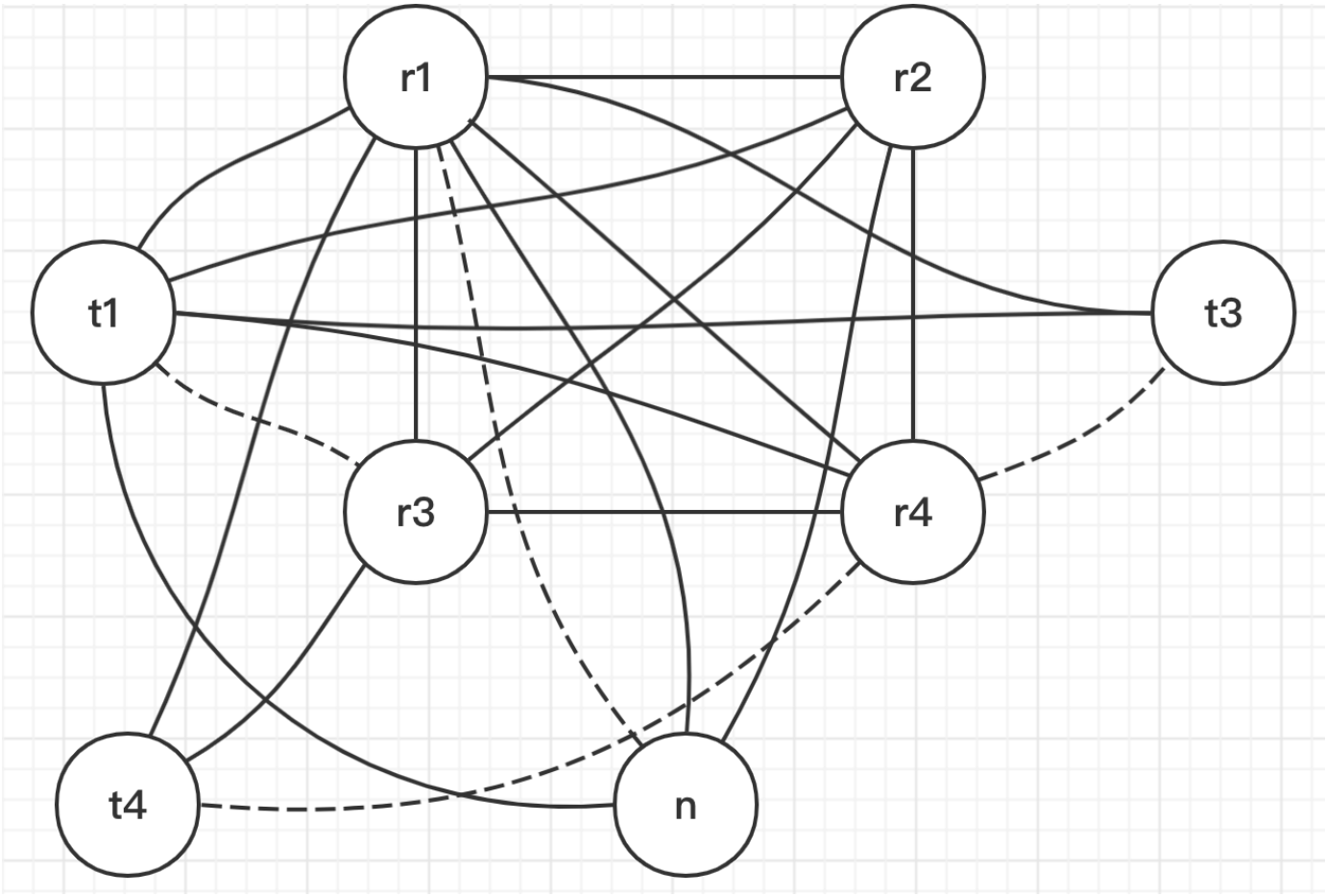
4. 冻结：如果 *简化* 和 *合并* 都不能再进行，就寻找一个 **度数较低**、**传送有关** 的结点，冻结这个结点所关联的所有 *传送指令*：放弃对那些传送指令进行合并的希望。**重新进行简化和合并阶段。**
5. 溢出：如果没有低度数结点，选择一个潜在可能溢出的高度数结点并将其压栈。
6. 选择：弹出整个栈并指派颜色。

e.g.

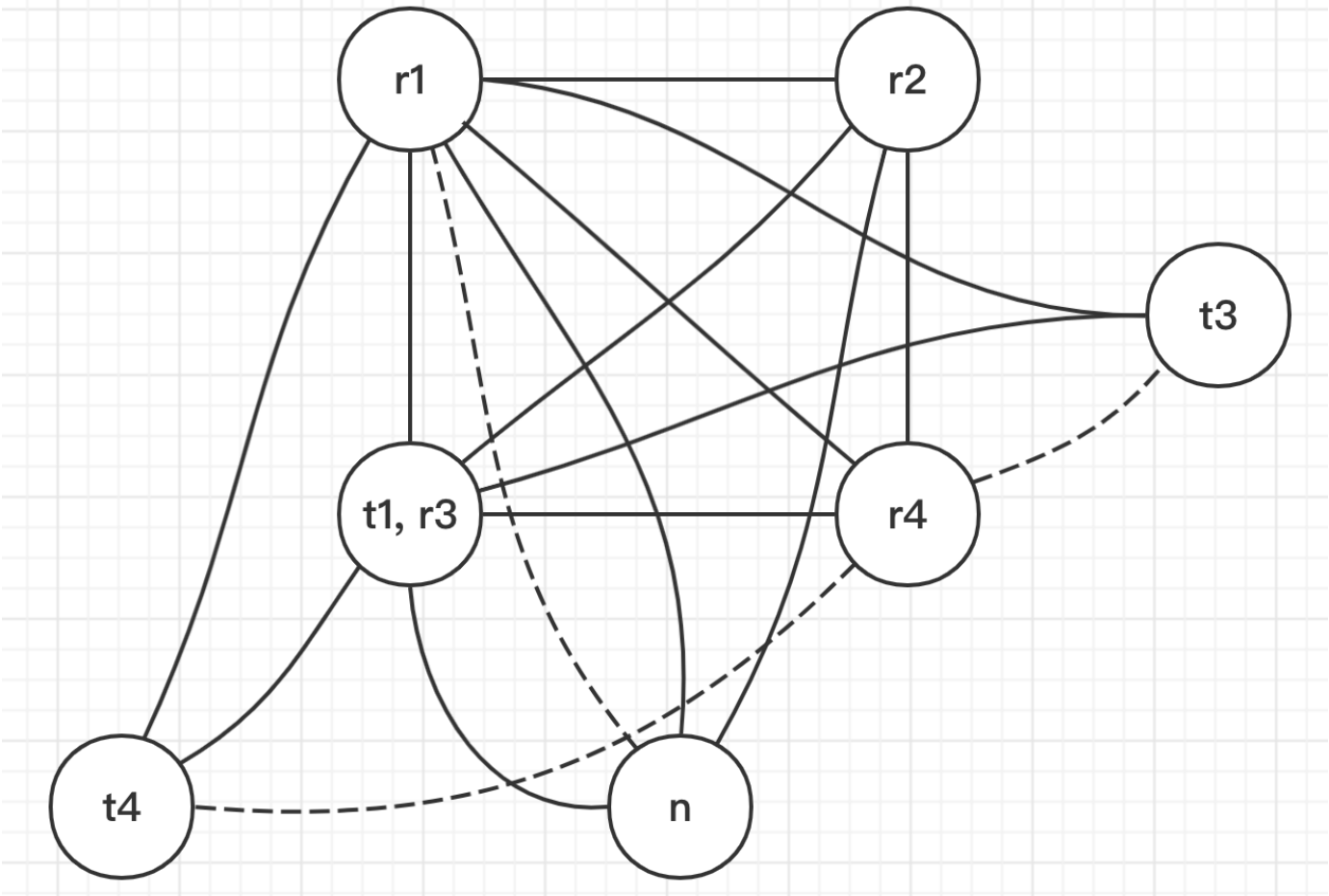
观察可知，原图很难简化和合并，所以我们选择溢出 $r4$ 并重写指令。

| Instr | def | use | in | out |
|---|---|---|---|---|
| t1 <- r3 | t1 | r3 | r1, r3, r4 | t1, r1, r4 |
| T3 <- r4 | t2 t3 | r4 | t1, r1, r4 | t1, r1, t3 |
| M[t2] <- t3 | | t3 | t1, r1, t3 | t1, r1 |
| n <- r1 | n | r1 | t1, r1 | t1, n |
| cmp n,1 | | n | t1, n | t1, n |
| jle end | | | t1, n | t1, n |
| r1 <- (n – 1) | r1 | n | t1, n | r1, t1, n |
| call nfactor | r1, r2 | r1 | r1, t1, n | r1, t1, n |
| r1 <- r1 * n | r1 | r1, n | r1, t1, n | t1, t1 |
| jmp ret | | | r1, t1 | r1, t1 |
| r1 <- 1 | r1 | | t1 | r1, t1 |
| r3 <- t1 | r3 | t1 | r1, t1 | r1, r3 |
| T4 <- M[t2] | t4 | | r1, r3 | r1, r3, r4 t4 |
| r4 <- t4 | r4 | t2 t4 | r1, r3, t4 | r1, r3, t4 r4 |
| return | | r1, r3, r4 | r1, r3, r4 | r1, r2, r4 r3 |

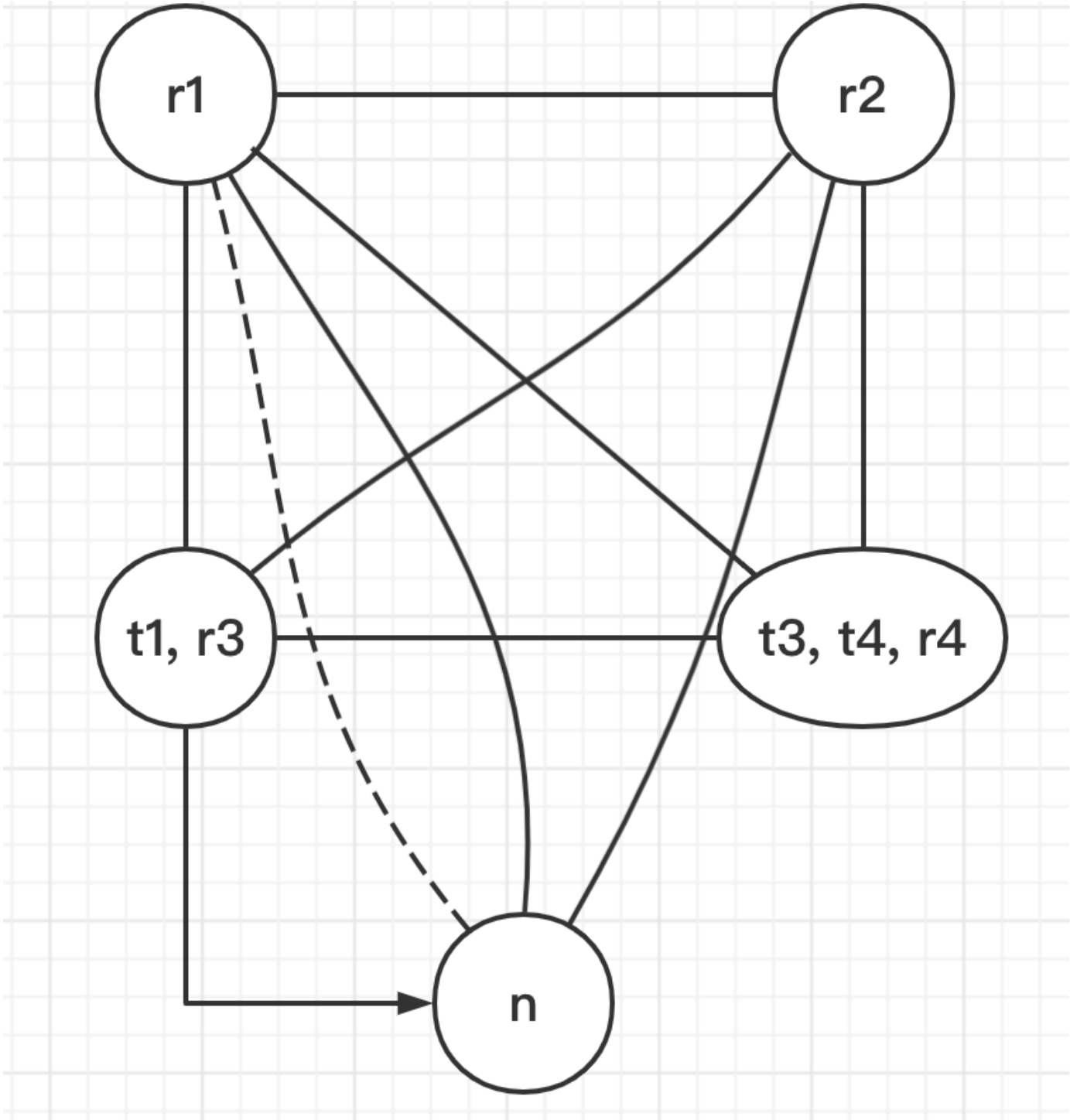
- 重新构造冲突图如下。




- 根据 Briggs 法则，选择传送指令相关的 $t1$ 和 $t3$ 结点进行合并，得到冲突图如下。



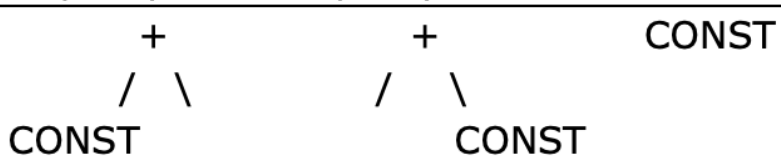
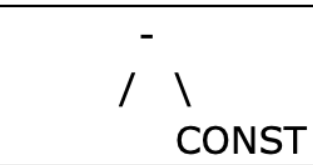
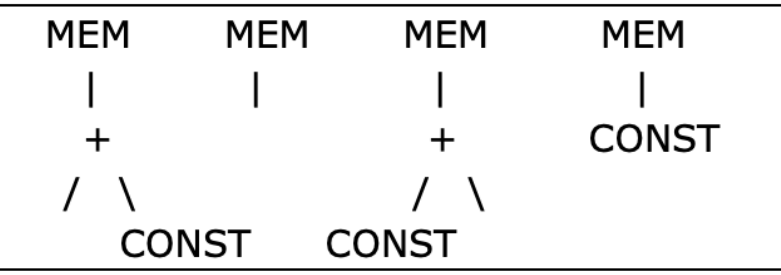
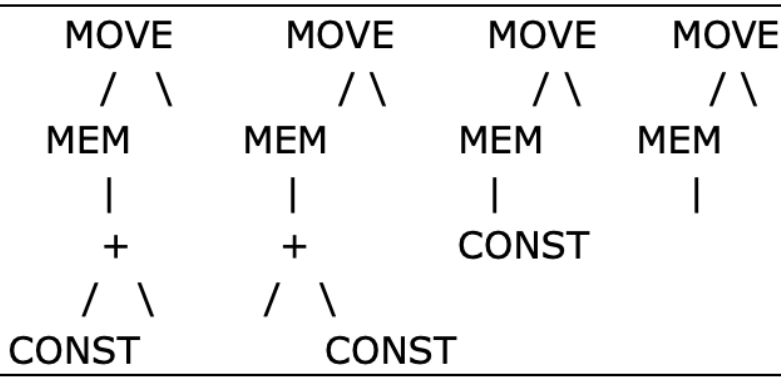
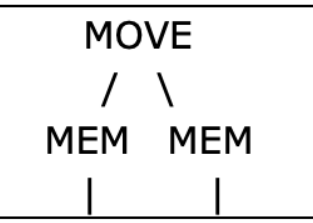

- 根据 Briggs 法则，选择传送指令相关的 $t3$ 和 $r4$、$t4$ 和 $r4$ 结点进行合并，得到冲突图如下。



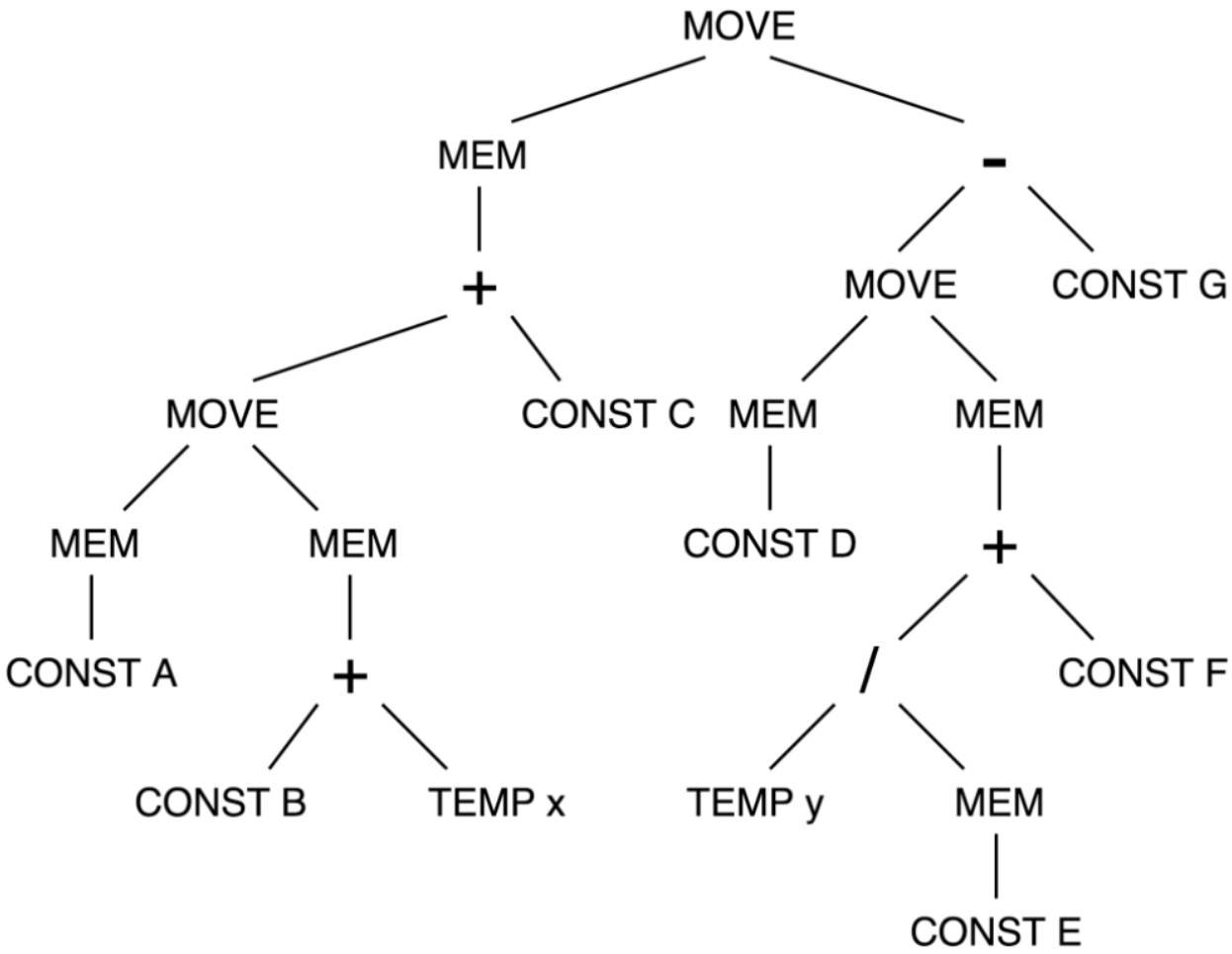

- 所有传送指令都是受抑制的，可以消除。
- 依次简化压栈 $n$，$r1$，$r2$，$t1$ & $r3$，$t3$ & $t4$ & $r4$。
- 分配 $r4$ 给 $n$，分配 $r3$ 给 $t1$，分配 $r4$ 给 $t3$ 和 $t4$。
- 染色成功，重写指令并消除多余传送指令，结果如下。

```
M[t2] <- r4
r4 <- r1
cmp r4, 1
jle end
r1 <- (r4 - 1)
call nfactor
r1 <- r1 * r4
jmp ret
r1 <- 1
r4 <- M[t2]
return
```

# 指令选择

| Name | Effect | Trees | Cost |
|---|---|---|---|
| - | $r_i$ | TEMP | 0 |
| ADD<br>MUL | $r_i \leftarrow r_j + r_k$<br>$r_i \leftarrow r_j * r_k$ |  | 1 |
| SUB<br>DIV | $r_i \leftarrow r_j - r_k$<br>$r_i \leftarrow r_j / r_k$ |  | 1 |
| ADDI | $r_i \leftarrow r_j + c$ |  | 1 |
| SUBI | $r_i \leftarrow r_j - c$ |  | 1 |
| LOAD | $r_i \leftarrow M[r_j + c]$ |  | 2 |
| STORE | $M[r_j + c] \leftarrow r_i$ |  | 2 |
| MOVEM | $M[r_j] \leftarrow M[r_i]$<br>$r_k \leftarrow M[r_i]$ |  | 4 |

Consider the following IR tree and answer the questions below.

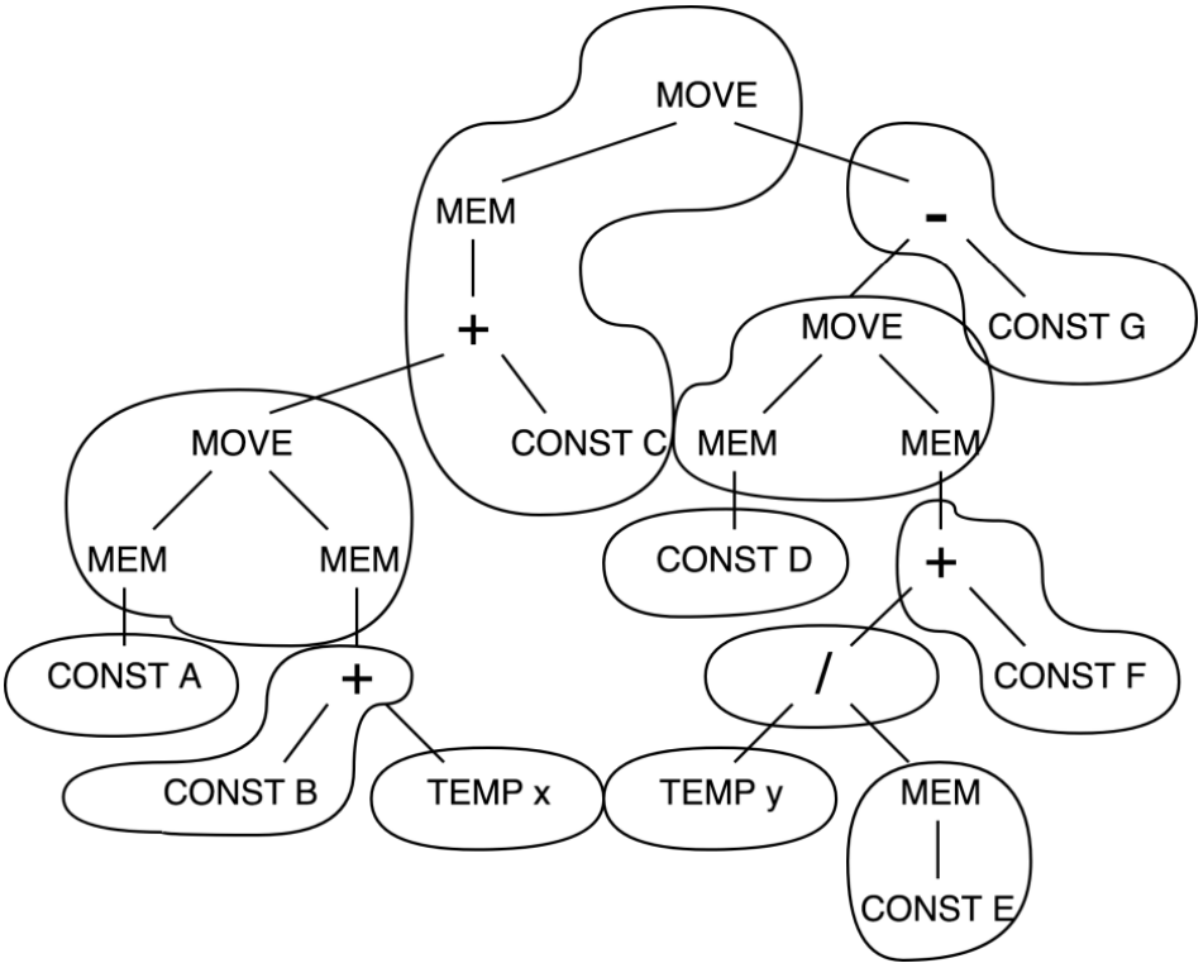



> 1. In the course, we have learnt **maximum munch algorithm** to tile the IR tree. Please use this algorithm to tile the IR tree above. **Draw your tiles, write the corresponding instruction sequences** after tiling, and **calculate the cost** of your instruction sequences. (10')

**Maximal Munch 算法（P138）**

从树的根节点开始，寻找适合它的 **最大瓦片（覆盖结点数最多的瓦片）**，用这个瓦片覆盖根节点，同时也可能会覆盖根节点附近的其他几个结点。覆盖根节点后，遗留下了若干子树，然后，对每一棵子树重复相同的算法。

**注：在 Jouette 体系结构中，寄存器 $r_0$ 总是包含 0。**




**COST=18**

再分块**后序遍历**来生成相应指令，原则就是从底向上，先生成出上层所需的内容。

```
ADD                    r1 <- r0 + A                          1
ADDI           r2 <- rx + B                          1
MOVEM   M[r1] <- M[r2]                  4
                               r2 <- M[r2]
ADD            r1 <- r0 + D                          1
LOAD    r3 <- M[r0 + E]                 2
DIV            r3 <- ry / r3                         1
ADDI           r3 <- r3 + F                          1
MOVE           M[r3] <- M[r1]                  4
                       r1 <- M[r1]
SUBI           r1 <- r1 - G                          1
STORE   M[r2 + C] <- r1                2
```

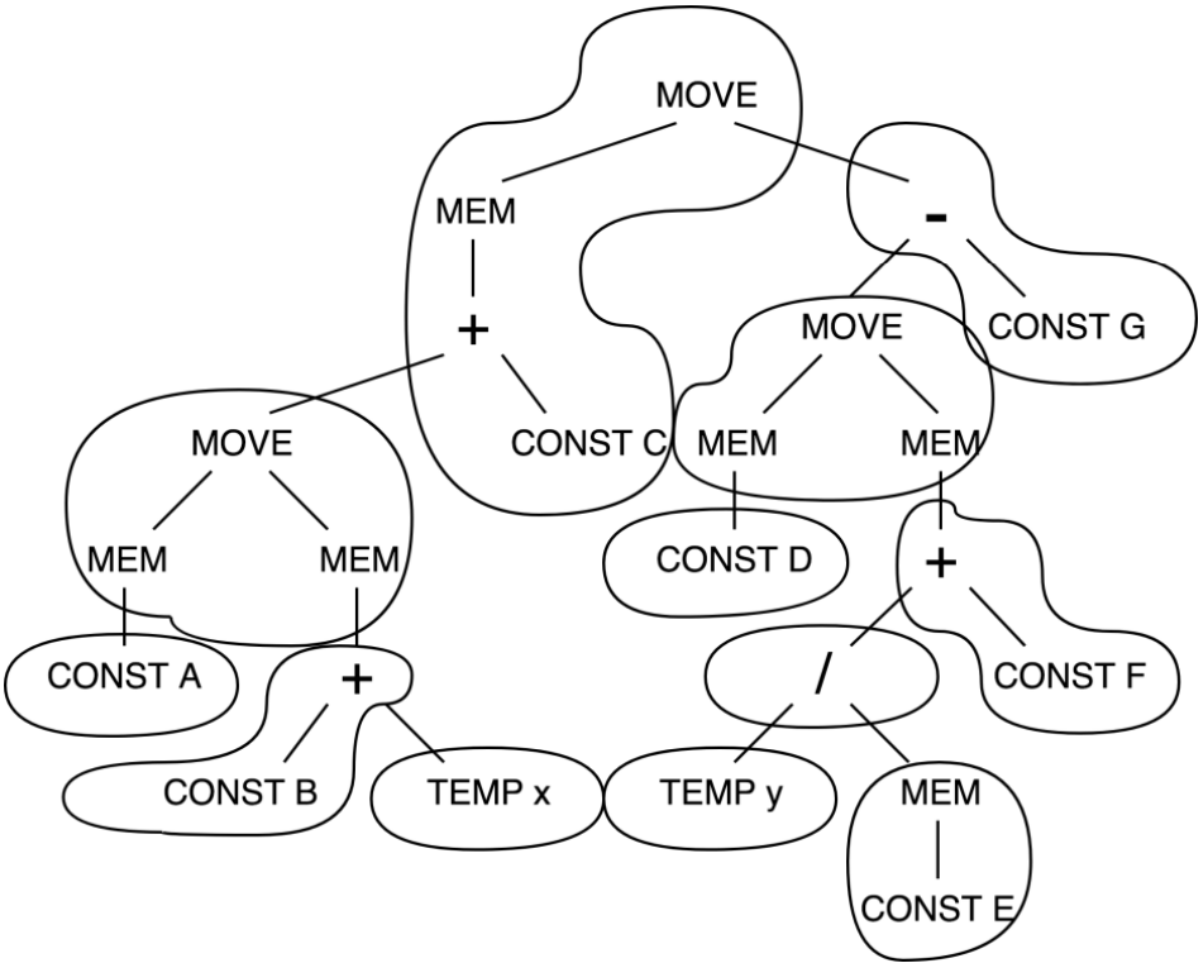
$$cost = 1 + 1 + 4 + 1 + 2 + 1 + 1 + 4 + 1 + 2 = 18$$

> 2. **Maximum munch algorithm** is an **optimal** tiling algorithm, and will not always generate tiles with smallest cost. A **dynamic-programming algorithm** can find the **optimum** tiling to the IR tree.
>
>    Please use the dynamic-programming algorithm to tile the IR tree above. **Draw your tiles, write the corresponding instruction**

> **sequences** after tiling, and **calculate the cost** of your instruction sequences. (15')

**动态规划算法（P140）**

自底向上，首先递归求出结点 $n$ 的所有儿子（和孙子）的代价，然后将每一种树型（瓦片种类）与结点 $n$ 进行匹配。




**COST=18**

```
ADD                    r1 <- r0 + A                          1
ADDI            r2 <- rx + B                          1
MOVEM   M[r1] <- M[r2]                  4
                                r2 <- M[r2]
ADD             r1 <- r0 + D                          1
LOAD    r3 <- M[r0 + E]                 2
DIV             r3 <- ry / r3                         1
ADDI            r3 <- r3 + F                          1
MOVE            M[r3] <- M[r1]                  4
                        r1 <- M[r1]
SUBI            r1 <- r1 - G                          1
STORE   M[r2 + C] <- r1                 2
```

$$cost = 1+1+4+1+2+1+1+4+1+2 = 18$$

# 垃圾收集

> 1. Compare copying collection with mark & sweep collection, what's the **pros & cons** of **copying collection?** (4')

- 优点：减少内存碎片化，操作简单复杂性低。
- 缺点：如果大部分变量存活时间很长，会导致内存拷贝过多，overhead 很大，而且复制回收只能用一半内存空间。

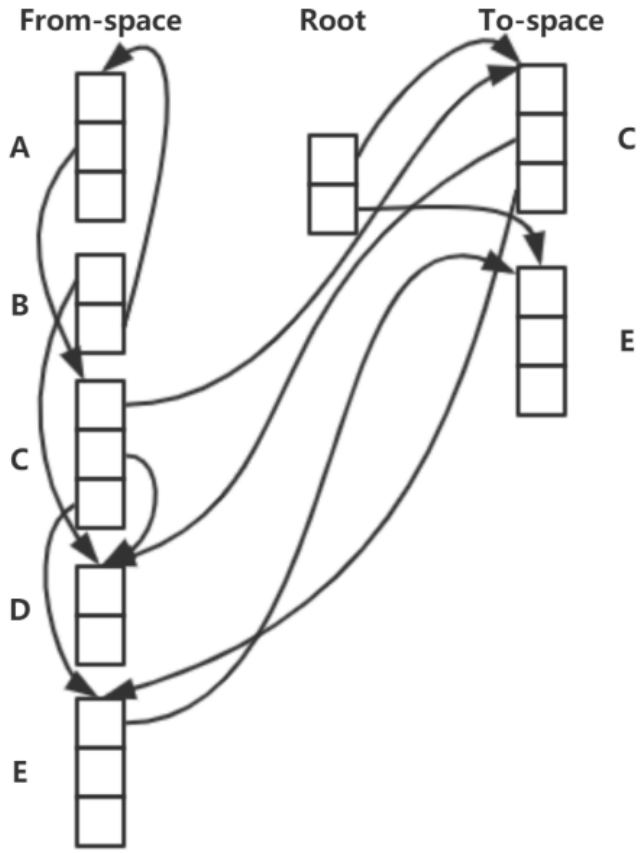
> 2. In the course, we have learnt that garbage collection help us recycle heap space occupied by non-live variables. How does the program reuse space in stack frame(spills in pre-allocated stack) occupied by non-live variables? (3')

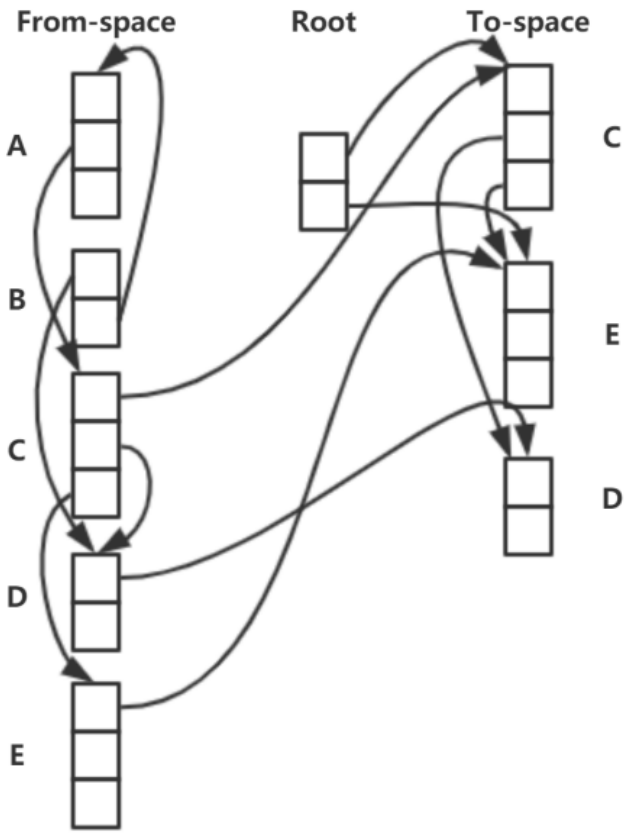
对溢出变量染色，进行内存分配，这样可以把活跃范围不重叠的溢出变量分配到同一个内存地址处。

> 3. When a garbage collection based on reachability analysis is about to happen, the program runtime usually scan stack frame and use reference variables in stack as roots. If a reference variable is not live, will we still use it as a root? Why? (3')

会，活性分析是编译时的工作，而垃圾回收是运行时做的，它无法获取到变量的活跃信息。

> 4. Use copying collection to finish the following datagram. You need to draw the memory state and pointer after forwarding roots and scanning. You can refer to figure 13-4 in the Chinese textbook or figure 13.10. in the English textbook (10')




## 一些概念

**垃圾**：在堆中分配且通过任何程序变量形成的指针链都无法到达的记录。

**垃圾收集**：垃圾占据的存储空间应当被回收，以便分配给新的记录的过程。

## 标记 - 清扫式收集

可参考文章

在标记清扫式中，程序变量和指针构成一个有向图，用 DFS 对图进行遍历，标记可以到达的点，于是未被标记的点就是垃圾，应当被回收。从第一个地址到最后一个地址对整个堆进行清扫。清扫出来的垃圾用一个链表（空闲表）链接在一起。

深度优先搜索

```
function DFS(x)
        if x 是一个指向堆的指针
                if 记录 x 还没有被标记
                        标记 x
                        for 记录 x 的每一个域 fi
                                DFS(x.fi)
```

标记-清扫式垃圾收集

标记阶段

```
        for 每一个根 v
                DFS(v)
```

清扫阶段

```
        p <- 堆中第一个地址
        while p < 堆中最后一个地址
                if 记录 p 已标记
                        去掉 p 的标记
                else 令 f1 为 p 中的第一个域
                        p.f1 <- freelist
                        freelist <- p
                        p <- p + (size of record p)
```

**假设在大小为 $H$ 的堆中有 $R$ 个字的可到达数据，则垃圾收集的分摊代价是：**

$$\frac{c_1R + c_2H}{H - R}$$

**使用一个显式的栈**

使用显式的栈而不是递归来实现深度优先搜索，但辅助栈的存储空间大小与被分配的堆空间大小相同仍然是不能接受的。

**指针逆转**

使用 $x.f_i$ 来存储栈自身的一个元素，之后在栈中弹出 $x.f_i$ 的内容时，再将域 $x.f_i$ 恢复为它原来的值，能够节约空间。

使用指针逆转的深度优先搜索

```
function DFS(x)
        if x 是一个指针并且记录 x 没有标记
                t <- nil
                标记 x; done[x] <- 0
                while true
                        i <- done[x]
                        if i < 记录 x 中域的个数
                                y <- x.fi
                                if y 是一个指针并且记录 y 没有标记
                                        x.fi <- t; t <- x; x <- y
                                        标记 x; done[x] <- 0
                                else
                                        done[x] <- i + 1
                        else
                                y <- x; x <- t
                                if x = nil then return
                                i <- done[x]
                                t <- x.fi; x.fi <- y
                                done[x] <- i + 1
```

### 空间表数组

使用一个由若干个空闲表组成的数组，使得 freelist[i] 是所有大小为 $i$ 的记录组成的链表。这样，当程序要分配一个大小为 $i$ 的结点时，只需取 freelist[i] 的表头即可。收集器的清扫阶段可以将每一个大小为 $j$ 的结点放在 freelist[j] 的表头处。

## 引用计数

可参考文章

编译器要生成一些额外的指令，使得每当将 $p$ 存储到 $x.f_i$ 时便增加 $p$ 的引用计数，并减少 $x.f_i$ 以前指向的记录的引用计数。如果某个记录 $r$ 的引用计数减少为零，则需将 $r$ 放到空闲表中，并且减少 $r$ 指向的所有其他记录的引用计数。

### 问题

1. 无法回收构成环的垃圾。
2. 增加引用计数所需的操作代价非常大。

### 解决方法

1. 简单要求程序员在使用一个数据结构时显示地解开所有的环。
2. 将引用计数（用于急切的且非破坏性的垃圾回收）与偶尔的标记-清扫（用于回收环）相结合。

## 复制式收集

可参考文章

遍历整个图（堆中称为 from-space 的部分），并在堆的新区域（称为 to-space)建立一个同构的副本。副本是紧凑的，它占据连续的、不含碎片的存储单元（即在可到达数据之间没有零散分布的空间记录）。原来指向 from-space 的所有的根在复制之后变成指向 to-space 副本，在此之后，整个 from-space （垃圾，加上以前可到达的图）便成为不可达到的。

### 收集的初始化

### 转递【P199】

### Cheney 算法【P200】

```
scan <- next <- to-space 的开始
for 每一个根 r
        r <- Forward(r)
        while scan < next
                for scan 处的那个记录的每一个域 fi
                        scan.fi <- Forward(scan.fi)
                scan <- scan + scan 处的那个记录的大小
```

Cheney 算法不需要外部的栈，也不需要逆转指针，它使用 scan 和 next 之间的 to-space 区域作为其宽度优先搜索队列，这使得它的实现比采用指针逆转的深度优先搜索简单得多。

**引用的局部性**

如果一个位于地址 $a$ 的记录指向另一个位于地址 $b$ 的记录，则 $a$ 和 $b$ 可能相距很远。

具有**良好的局部引用性**非常重要，在深度优先遍历中父亲节点往往与其孩子节点相邻较近，故其局部引用性较好。

**垃圾收集的代价**

$$\frac{c_3 R}{\frac{H}{2} - R}$$

```
function Forward(p)
        if p 指向 from-space
                then if p.f1 指向 to-space
                        then return p.f1
                        else Chase(p); return p.f1
                else return p

function Chase(p)
        repeat
                q <- next
                next <- next + 记录 p 的大小
                r <- nil
                for 记录 p 的每一个域 fi
                        q.fi <- p.fi
                        if q.fi 指向 from-space 且 q.fi.f1 不指向 to-space
                        then r <- q.fi
                p.f1 <- q
                p <- r
        until p = nil
```

# 分代收集

可参考文章

为了避免在所有的 $G_1, G_2, \cdots$，中搜索 $G_0$ 的各个根节点，我们让编译好的程序记住何处存在有这种从老对象指向新对象的指针。有以下几种方法：【P202】

根据对象的活跃度将对象氛围几个“代（G）”，年轻的会慢慢变老哦。收集器会更多得关注年轻的代（其成为垃圾的可能性更高）。但年老的指向年轻的，这样光对年轻代扫描得到的数据不准确。

- 记忆表：用向量记录更新过的对象
- 记忆集合：用对象内的一位来记录是否更新过
- 卡片标记：划分存储区

- 页标记：利用操作系统中的**赃位**。

**分代收集的代价**

## 增量式收集

可参考文章

提供了良好的交互性。

- 栅栏写：每一条**存数指令**进行检查以确保其遵守相关不变式。
- 栅栏读：每一条**读数指令**进行检查以确保其遵守相关不变式。

**三色标记**：在标记清扫式或复制式垃圾收集方法中，有以下三种记录：

- **白色** 对象是用深度优先或宽度优先搜索那些还未访问过的对象。
- **灰色** 对象是那些已经被访问过（标记或复制），但其儿子还未被查看过的对象。
- **黑色** 对象是那些已经被标记过，并且其儿子也已被标记过的对象。

```
while 存在任何灰色对象
        选择一个灰色记录 p
        for p 的每一个域 fi
                if 记录 p.fi 是白色
                        将记录 p.fi 涂成灰色
        将记录 p 涂成黑色
```

## Baker 算法

基于 Cheney 复制式收集算法，可与分代收集兼容。

Baker 算法的最大代价是为了维持不变式而在每条取数指令之后增加的额外指令。

# 函数式语言程序设计

## 闭包

闭包是一个记录，包含了指向函数机器代码的指针及访问必须的非局部变量的途径。一种简单的闭包可以只包含代码指针和静态链，非局部变量可以通过这个静态链来访问。

**环境**：闭包中给出对变量值的访问途径的部分通常称为环境。

## 堆上分配的活动记录