Compiler
2018 Fall Final Examination

Name_____ Student No._____ Score_____

Problem 1: (15 points)

1 (7′ 一个 Env 写错扣 2 分，意思对即可)
只写每次变化的

E0={a->int, b->int}

E1={t->string,res->int}

E2={t->int}

E3={t->string}

或每次可见的均可

E0={a->int,b->int}

E1={a->int,b->int,t->string,res->int}

E2={a->int,b->int,t->int,res->int}

E3={a->int,b->int,t->string,res->int } or E3 = E1

其他表达相同意思的均可

2 (8′ 一个点 2 分，意思对即可)

2: E0

6: E1

11-18:E2

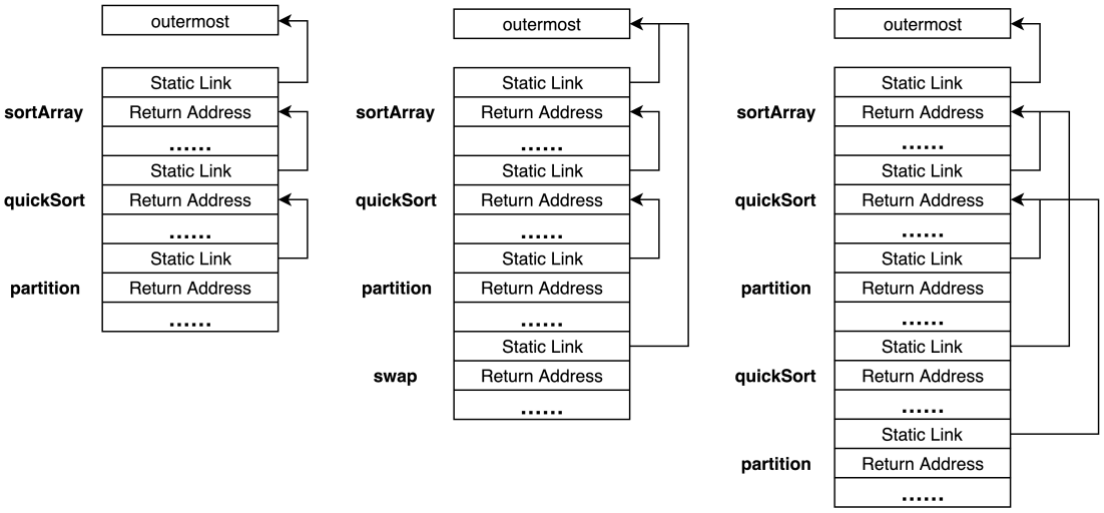
20:E3

其他表达相同意思的答案均可

Problem 2: (20 points)

1 (10′=3′+3′+4′，意思对即可)

(1)

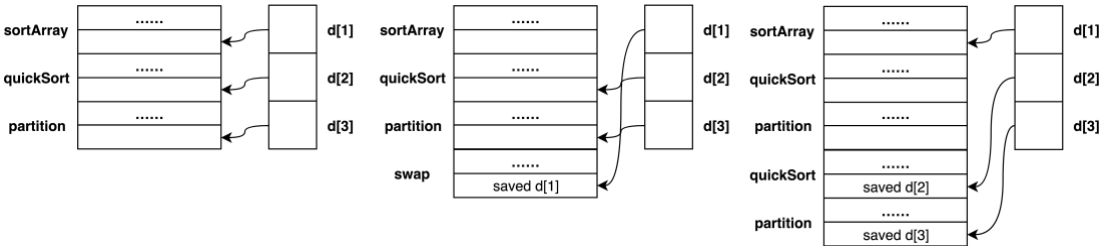


(2)

(3)

2 （10′=3′+3′+4′，意思对即可）

(1)

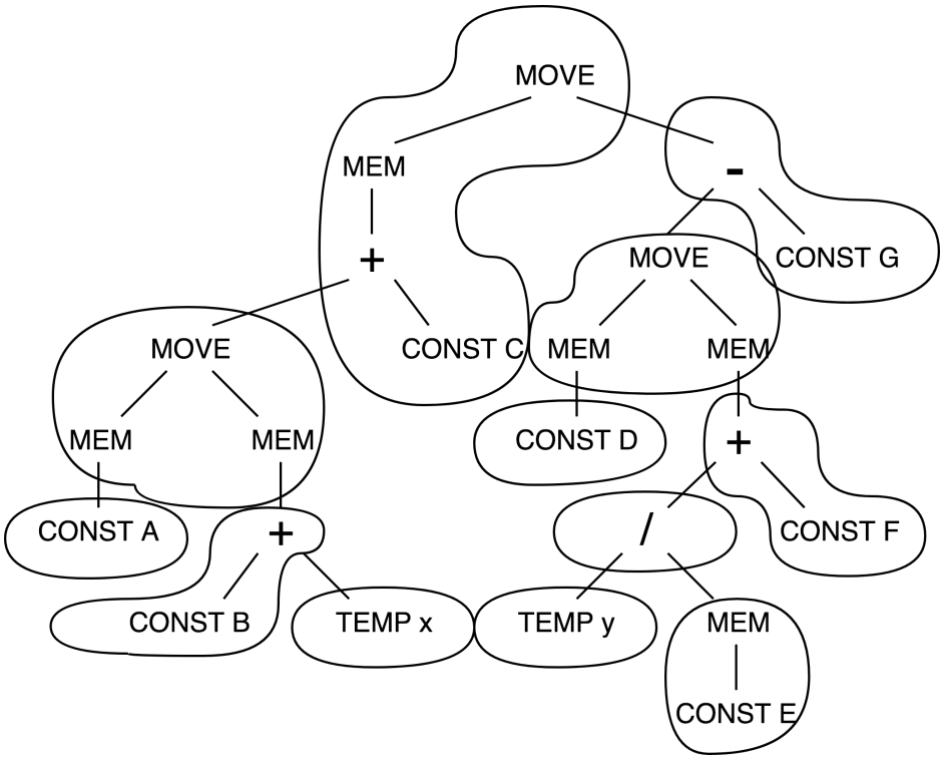


(2)

(3)

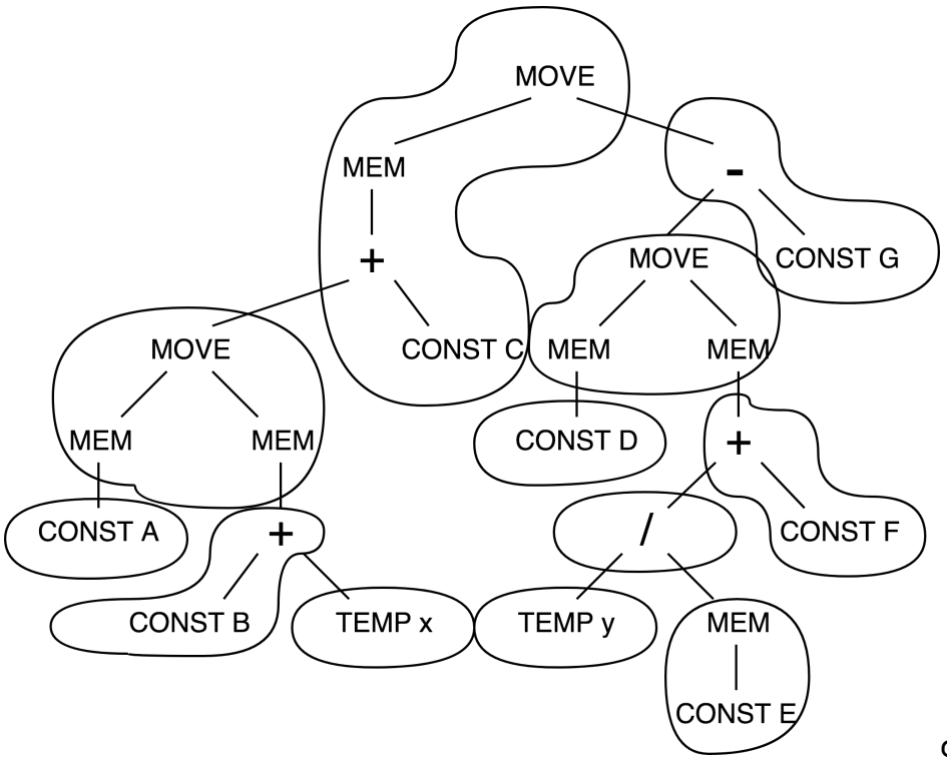
Problem 3: (25 points)

1. (10′，4 分 munch 正确性，4 分指令，2 分计算 COST)



COST=18

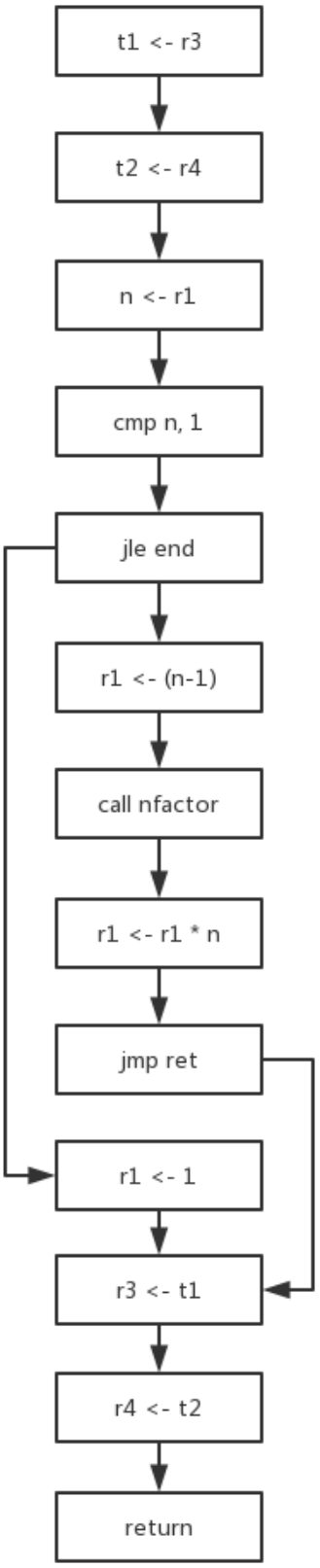
2 (15′，6 分 munch 正确性，6 分指令，3 分计算 COST)



COST=18

Problem 4: (40 points)

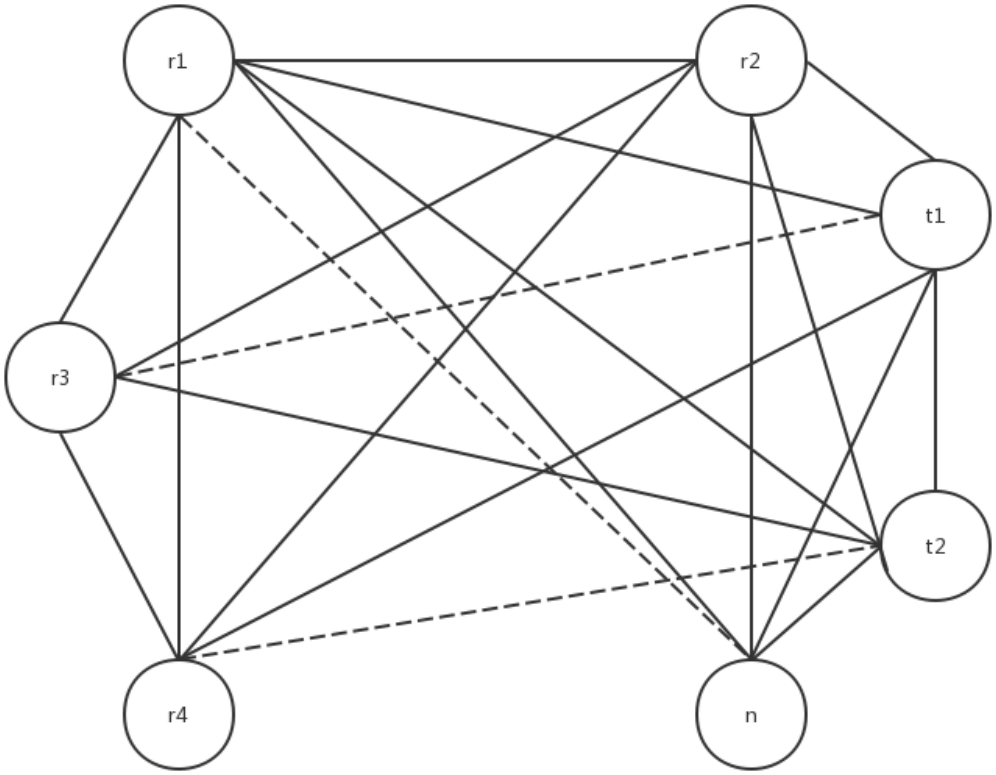
1 (5′ CFG 的正确性)



2. (10’Use-Def 正确性，活跃分析正确性，重点关注 caller-saved 寄存器)

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

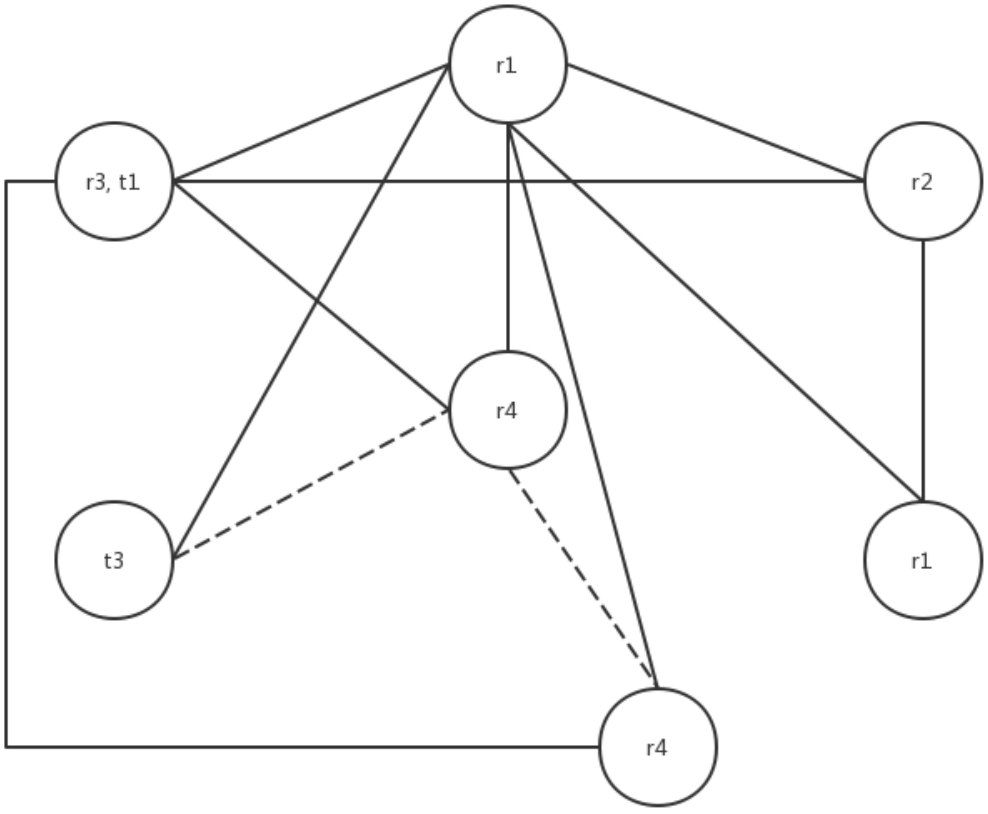
3. (10’ 冲突图的正确性)



4. (15' rewrite 后的正确性，spill 的正确性)

每个部分各 5 分

Instr	def	use	in	out
t1 <- r3	t1	r3	r1, r3, r4	t1, r1, r4
T3 <- r4	t2	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
r4 <- t4	r4	t2	r1, r3, t4	r1, r3, t4
return		r1, r3, r4	r1, r3, r4	r1, r2, r4



```
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
```

Problem 1: Type Checking (15 points)

```
1  function isRP(a:int, b:int) : int = (
2     print_int(a); print_int(b);
3     let
4        var t := "If relatively prime: "
5        var res := 0
6     in
7        if a = 1 & b = 1 then res := 1
8        else (
9           let
10             var t := 0
11          in
12             while 1
13             do (
14                t := a - a / b * b;  /* i.e., a % b */
15                if t = 0 then break
16                else (a := b; b := t)
17             );
18             if b > 1 then res := 0
19             else res := 1
20          end);
21       print(t); print_int(res); res
22    end
23 )
```

In tiger compiler's semantic analysis phase, we use **symbol tables** (also called **environments**) mapping identifiers to their types. An environment is a set of bindings denoted by the → arrow. For example, **E** = {g → string, a → int} means that in environment **E**, the identifier ***a*** is an ***integer*** variable and ***g*** is a ***string*** variable.

1. How many **environments** will be generated in the semantic analysis phase of the function ***isRP***? Please write them in detail. (You can answer like this: E0 = {id1→type1},E1={id2→type2, id3→type3},…) (7')

2. Each local variable in a program has a **scope** in which it is visible. Through looking into the **active environments**, the compiler can check whether the symbol exists. For the above function ***isRP***, you have written the generated **environments** in **Question1**. Please write all **active environments** for the following lines:**2,6,11-18,20** in ***isRP***. (You can answer like this: **'11-18:E0,E1**'). (8')

Problem 2: Static Link (25 points)

```
let
    type IntArray = array of int
    var len := 16
    var arr := IntArray [len] of 0

    function swap(a:int, b:int) =
      let
          var t := arr[a]
      in
          arr[a] := arr[b]
          arr[b] := t
      end

    function sortArray(len:int) =
      let
          var head := 0
          var tail := len - 1
          function quickSort(low:int, high:int) =
            let
                var pi := 0
                function partition(low:int, high:int) : int =
                  let
                      var pivot := arr[high]
                      var i := low - 1
                  in
                      for j := low to (high - 1)
                          do if arr[j] <= pivot then
                          (i := i + 1; swap(i, j))
                      swap(i + 1, high); i + 1
                  end
            in
                if low < high then
                (pi := partition(low, high);
                quickSort(low, pi - 1);
                quickSort(pi, high))
            end
      in
          quicksort(head, tail)
      end
in
    sortArray(len)
end
```

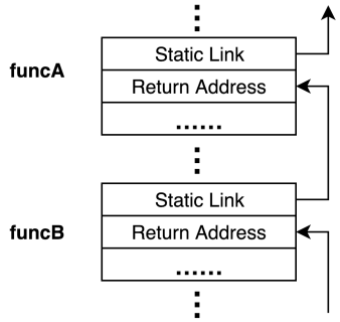
1. Suppose that we implement the nested functions using an access link. Please draw the access link in each of the activation records on the stack based on the following function invocations (10')

   1) sortArray→ quickSort→ partition

   2) sortArray→ quickSort→ partition→ swap

   3) sortArray→ quickSort→ partition→ swap→ quickSort→ partition

   e.g.



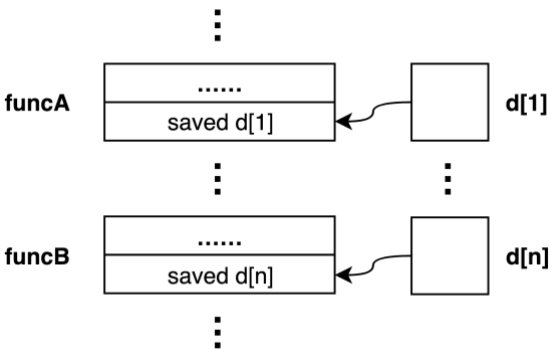
2. Suppose that we implement nested functions using displays. Please draw the display in each of the activation records on the stack based on the following function invocations. (15')

   1) sortArray→ quickSort→ partition

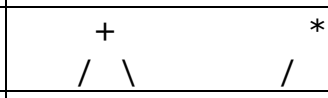
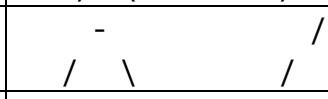
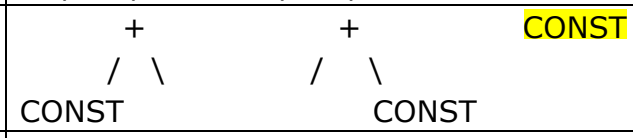
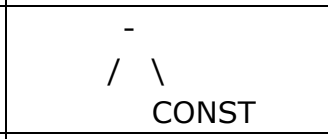
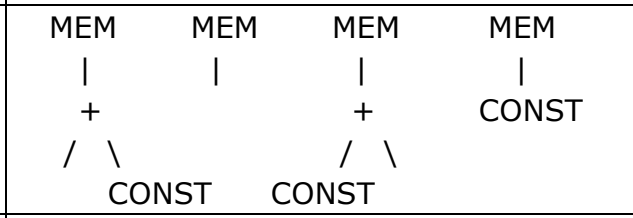
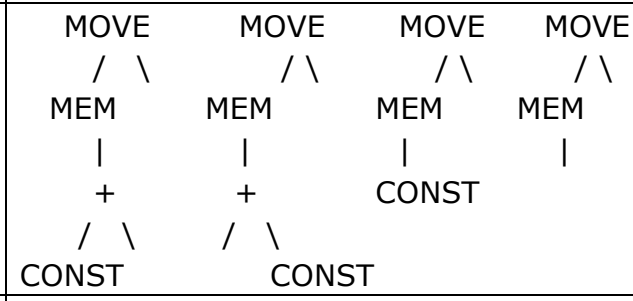
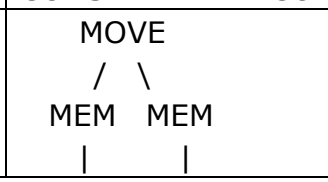
   2) sortArray→ quickSort→ partition→ swap

   3) sortArray→ quickSort→ partition→ swap→ quickSort→ partition

   e.g.



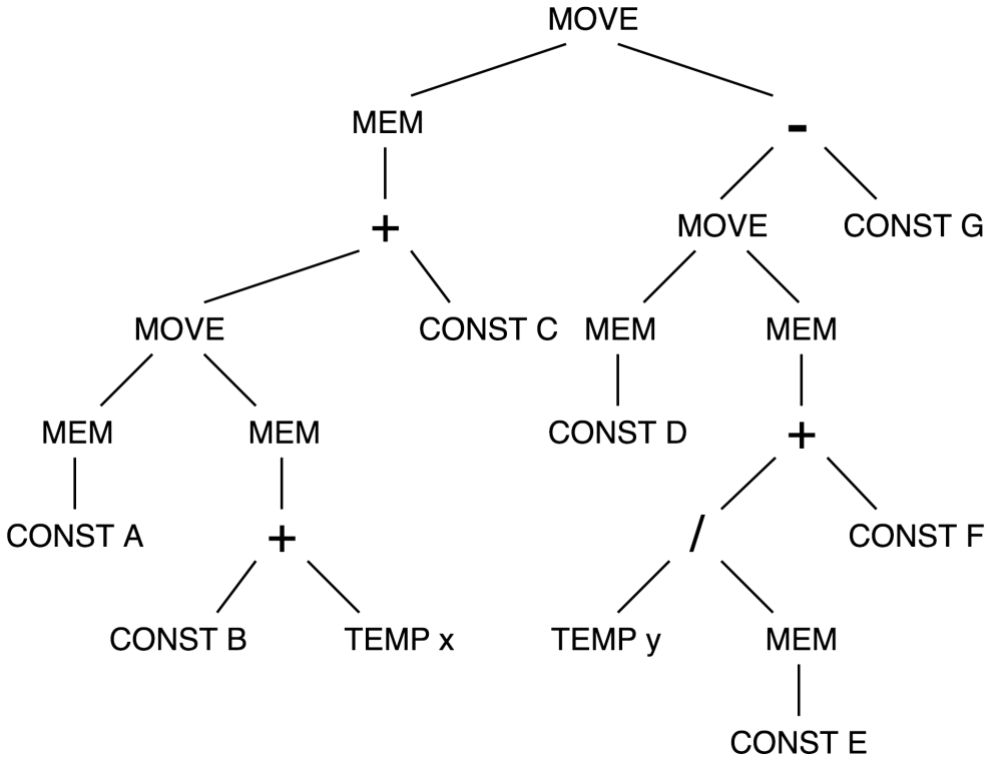
Problem 3: Instruction Selection (25 points)

Name	Effect	Trees	Cost
-	r_i	TEMP	0
ADD MUL	$r_i \leftarrow r_j + r_k$ $r_i \leftarrow r_j * r_k$		1
SUB DIV	$r_i \leftarrow r_j - r_k$ $r_i \leftarrow r_j / r_k$		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	$M[r_j] \leftarrow M[r_i]$ $r_k \leftarrow M[r_i]$		4

Note:

- The notation ***M[x]*** denotes the memory word at address x.
- The ***third column*** in the table above is the set of ***tiles*** corresponding to the machine instruction in the first column of the table.
- The ***fourth column*** in the table above is the ***cost*** corresponding to the machine instruction in the first column of the table.

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')

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')

Problem 4: Register allocation (40 points)

Suppose a compiler has compiled the following function **nfactor** into instructions on the right:

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 are **caller-saved** while r3, r4 are **callee-saved**.
- The **return value** will be stored in r1.
- t1, t2, n are temporary registers

Please answer the following questions according to the instructions.

1. Draw a control flow graph instruction-by-instruction. (5’)
2. Fill up the following def/use/in/out chart. (10’)

Instr	def	use	in	out
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				
return				

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

4. Adopt the graph-coloring algorithm to allocate registers for temporaries. (15')
   a) If there exists any spilling:
      i. rewrite the instructions and fill the chart below.

Instr	def	use	in	out

      ii. Draw the new interference graph according to the chart.
   b) Write down the final instructions after register allocation.

NOTE: Please make sure that you write the procedure clearly. Otherwise, you can get part of scores if you just provide the final instructions.