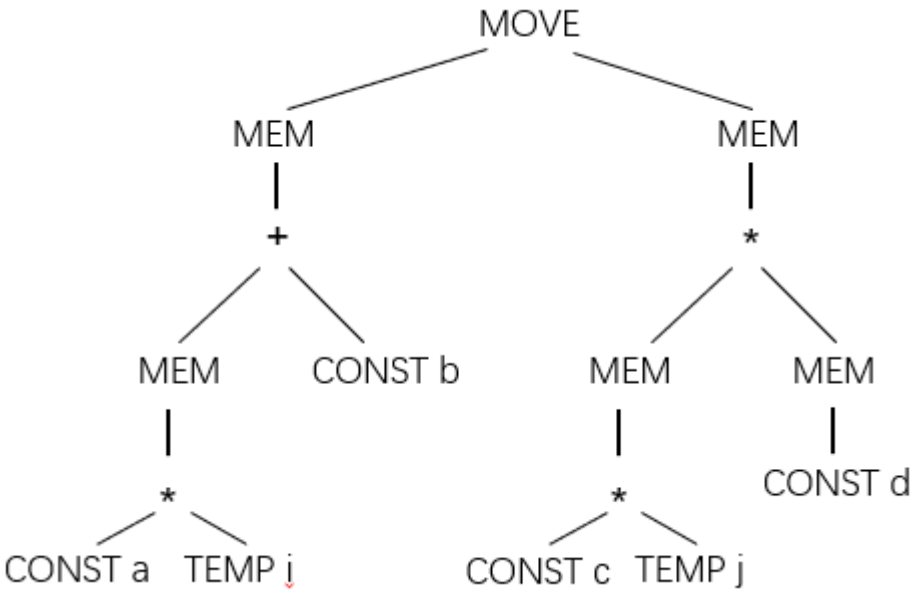
Problem 1: Instruction Selection (24 points)

Name	Effect	Trees	Cost
-	Ri	TEMP	0
ADD SUB	ri ← rj + rk ri ← rj - rk	+ - / \ / \	1
MUL DIV	ri ← rj * rk ri ← rj / rk	* / / \ / \	2
ADDI	ri ← rj + c	+ + CONST / \ / \ CONST CONST	1
LOAD	ri ← M[rj + c]	MEM MEM MEM MEM | | | | + + CONST / \ / \ CONST CONST	3
LOADX	ri ← M[rj * c]	MEM MEM | | * * / \ / \ CONST CONST	4
STORE	M[rj + c] ← ri	MOVE MOVE MOVE MOVE / \ / \ / \ / \ MEM MEM MEM MEM | | | | + + CONST / \ / \ CONST CONST	3
MOVEM	M[rj] ← M[ri]	MOVE / \ MEM MEM | |	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
- On this architecture, a register r_0 always contains zero.

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. (12')

2. We also have learnt a **dynamic programming algorithm** to 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. (12')

Problem 2: Garbage Collection (24 points)

```
 1  InitialMark:
 2    Stop Mutator
 3    for r in GCRoots:
 4      Mark(*r)
 5    end for
 6    Start Mutator
 7
 8  ConcurrentMark:
 9    while stack is not empty:
10      reference <- stack.pop
11      object <- *reference
12      Mark(object)
13    end while
14
```

```
15  Mark(object)
16    if object is not null and object.color is not black:
17      for field in object:
18        if field is a pointer and *field is not null:
19          ref <- *field
20          ref.color <- grey
21          stack.push(field)
22        end if
23      end for
24      object.color <- black
25  end for
26
27  GC
28    InitialMark
29    ConcurrentMark
30    Remark
31    Colletion
```

Reading the above marking algorithm, try to answer the following questions(Note: At the beginning of GC, color of each object is white and stack is empty.)

1. What's the pros and cons of Concurrent Collection compared with Stop-The-World Collection? (4')
2. Clearly, this algorithm cannot handle the newly-allocated objects, suppose we could collect all newly-allocated objects after InitialMark in a set named New, design the Remark Phase and color all newly-allocated objects black. (4')
3. After Q2, does every live object is marked as black? If not, point out the problem of this algorithm, and try to fix it through designing a write barrier and completing the Remark phase you design in Q2. (8')

Write barrier work like this:

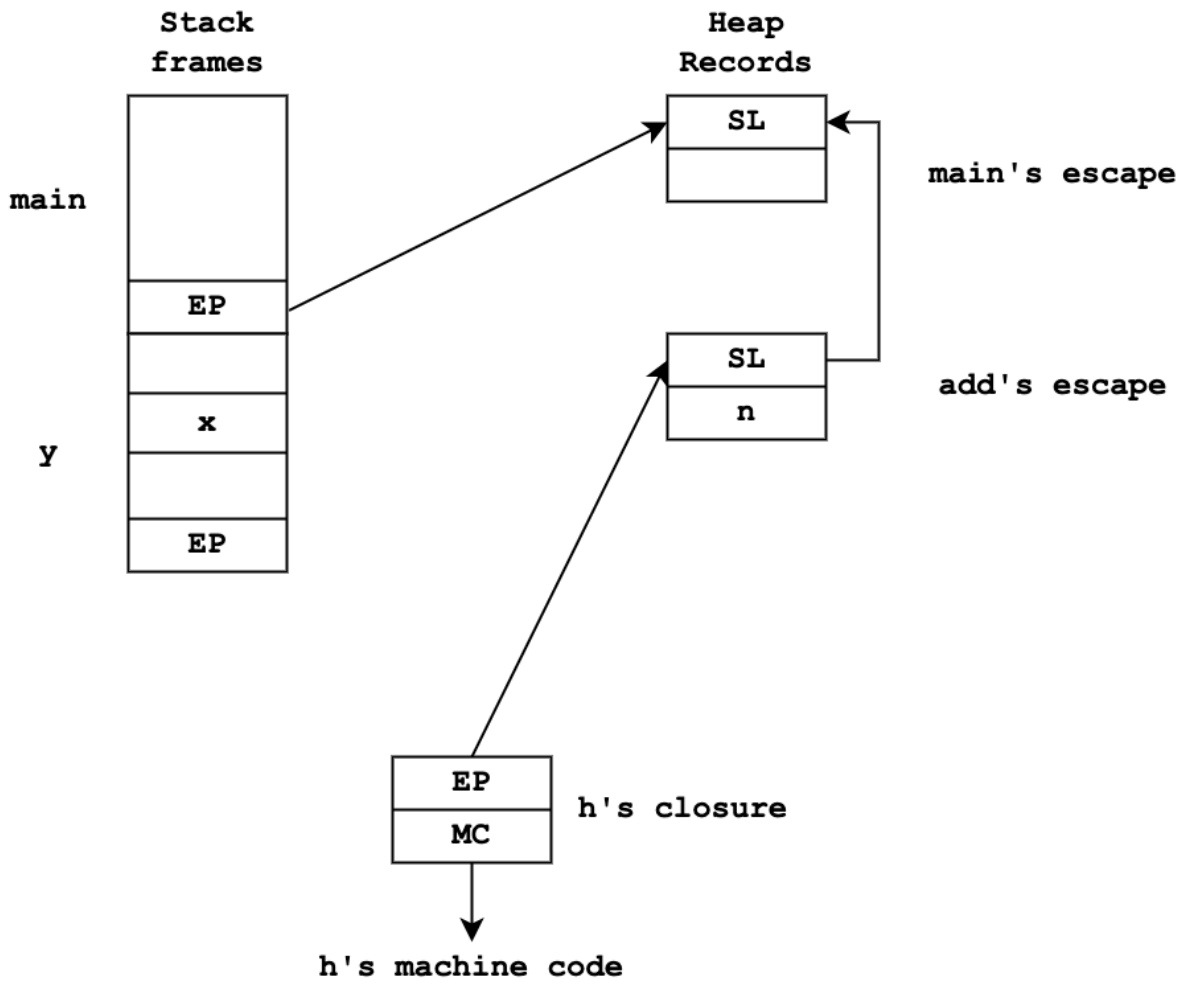
```
Putfield(obj, field, val)
  obj.field <- val
  if field is a pointer:
    WriteBarrier(obj, obj.field)
  end if
```

4. Floating garbage is dead objects that survive after GC. After Q3, this algorithm may still produce floating garbage, try to point out the problem that produces floating garbage and analyze whether it's possible to fix this based on the current mechanism. (You don't need to give the solution, just analyze whether we could fix this or not) (8')

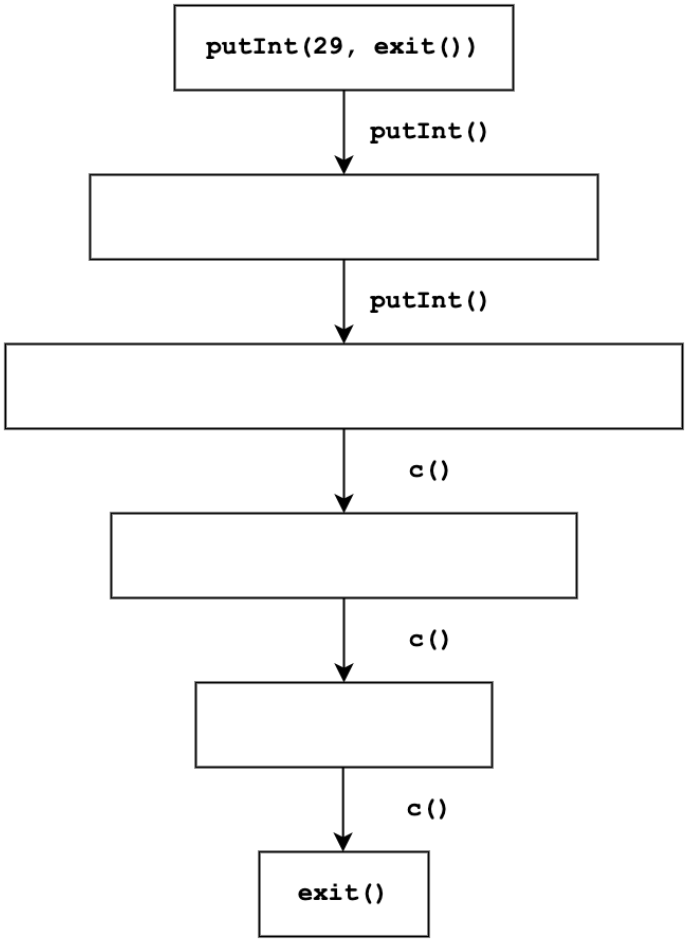
Problem 3: Functional Programming (20 points)

```
1   let
2       type intfun = int -> int
3
4       function add(n: int) : intfun =
5           let function h(m: int) : int = n+m
6            in h
7           end
8
9       function twice(f: intfun) : intfun =
10          let function g(x: int) : int = f(f(x))
11           in g
12          end
13
14      var x := twice(add(5))(7)
15      var y := twice(add(6))
16
17      function putInt(i: int, c: cont) =
18          if i=0 then c()
19          else let var rest := i / 10
20                   var dig := i - rest * 10
21                   function doDigit() = print(chr(dig), c)
22               in putInt(rest, doDigit)
23              end
24
25   in
26      var result = y(x)
27      putInt(result, exit)
28  end
```

1. When function **y** is invoked in **line 26**, part of the allocated activation records is shown in the following figure. Please draw the missing activation record(s) and add arrows to indicate the relation of different activation records. (8’)



2. What's the actual value of **x** and **n** in the figure? (4')
3. Please fill up the following control flow graph for all the function calls invoked within **line 27**. Every blank should be filled with the current function call with the actual value of its arguments. Hint: You should extract the stringed function calls in a continuation argument. (8')



Problem 4: Register Allocation (32 points)

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

```
int deadloop_search(int i) {
    int res = compare(i);
    if (res)
        return deadloop_search(i - 1);
    else
        return deadloop_search(i + 1);
}
```

```
deadloop_search:
    t1 <- r1
    t2 <- r2
    i <- r3
    call compare
    res <- r5
    cmp res, 0
    je L2
L1:
    r3 <- i
    r3 <- r3 - 1
    j L3
L2:
    r3 <- i
    r3 <- r3 + 1
```

	L3: call deadloop_search r2 <- t2 r1 <- t1 ret

Several things are worth noting in the instructions:

- There are **five** hardware registers in all.
- The compiler passes parameters using registers. (The first parameter goes to **r3**, the second goes to **r4**)
- **r1,r2** is **callee-saved** while **r3,r4** are **caller-saved**.
- The return value will be stored in **r5**.
- **t1, t2, i, res** are temporary registers.

Please answer the following questions according to the instructions.

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

2. Fill up the following def/use/in/out chart. (8')

instr	def	use	in	out
t1 <- r1				
t2 <- r2				
i <- r3				
call compare				
res <- r5				
cmp res, 0				
je L2				
r3 <- i				
r3 <- r3 - 1				
j L3				
r3 <- i				
r3 <- r3 + 1				
call deadloop_search				

`r2 <- t2`				
`r1 <- t1`				
`ret`				

3. Draw the interference graph for the program. Please use dashed lines for move edges and solid line for real interference edges. (4’)

4. The heuristic to decide which temporary to spill is the same as that in the Tiger book. i.e. The spill priority is calculated as:

```
Spill Priority =
(Uses+Defs-outside-loop + Uses+Defs-within-loop * 10) / Degree
```

Adopt the graph-coloring algorithm to allocate registers for temporaries. Write down the instructions after register allocation. (16’)

NOTE: You must write down **every decision you make** such as simplifying, spilling, coalescing. And you also need to write down **new instructions** and draw **new interference graph after each spilling**. If you answer is too simple, you will get a part of score.