**CS131 Final Exam Cover Sheet (Solution)**

Instructor: Fu Song

January 2, 2019

Your name: Your email ID: @shanghaitech.edu.cn

**INSTRUCTIONS**

* You have 150 minutes (09:00-11:30) to complete the exam.
* Your exam will not be graded unless you complete the cover sheet, and turn in both this exam book and the cover sheet.
* This exam is open-book and open-notes. You **cannot** use laptops, phones and e-readers to read electronic notes, or access to the internet for any reason.
* Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

Write on the following lines:

*“I certify that I am the person with the above name and email address. By taking this exam, I agree to all the rules set forth by the instructor.”*

SIGN your name:

|  |  |  |
| --- | --- | --- |
| Problem | Max | Points |
| 1 | 30 |  |
| 2 | 20 |  |
| 3 | 10 |  |
| 4 | 15 |  |
| 5 | 10 |  |
| 6 | 15 |  |
| Total | 100 |  |

You must turn in both the cover sheet and exam book.

**Question 1 (10+10+10=30 Points).** Consider the following subset of Cool expressions, extended with a **for** expression:

E🡪 **for** id = E **to** E **do** E

| **let** id: Type 🡨 E **in** E

| id 🡨 E

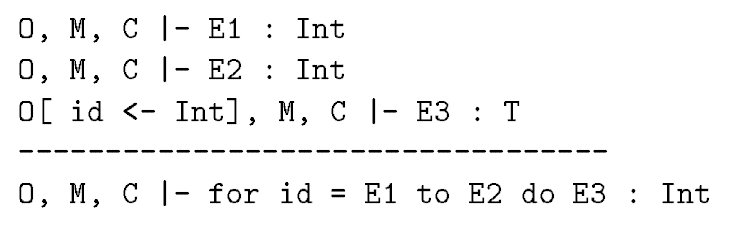
| **int**

In this grammar, **id** is a variable name, **Type** is a Cool class name. The semantics of

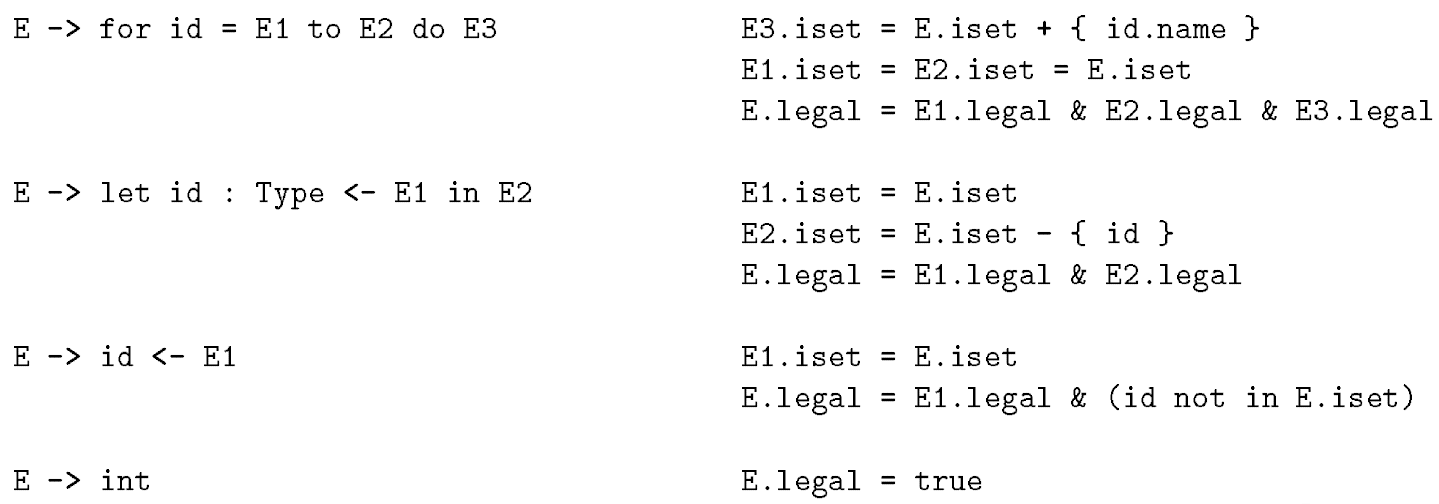
**for** id = E1 **to** E2 **do** E3 are

*E1* is evaluated giving an integer *i*, *E2* is evaluated giving an integer *j*, and then *E3* is evaluated with id =x for each *i <=x <= j* in order. If *i>j*, then the **for** expression terminates without evaluating *E3*. The result of the **for** expression is 0. The id is a new variable that hides any definitions in outer scopes; the scope of id is *E3*.

**Part (a).** Give a type rule for the **for** expression. Your rule should be as accurate as possible.



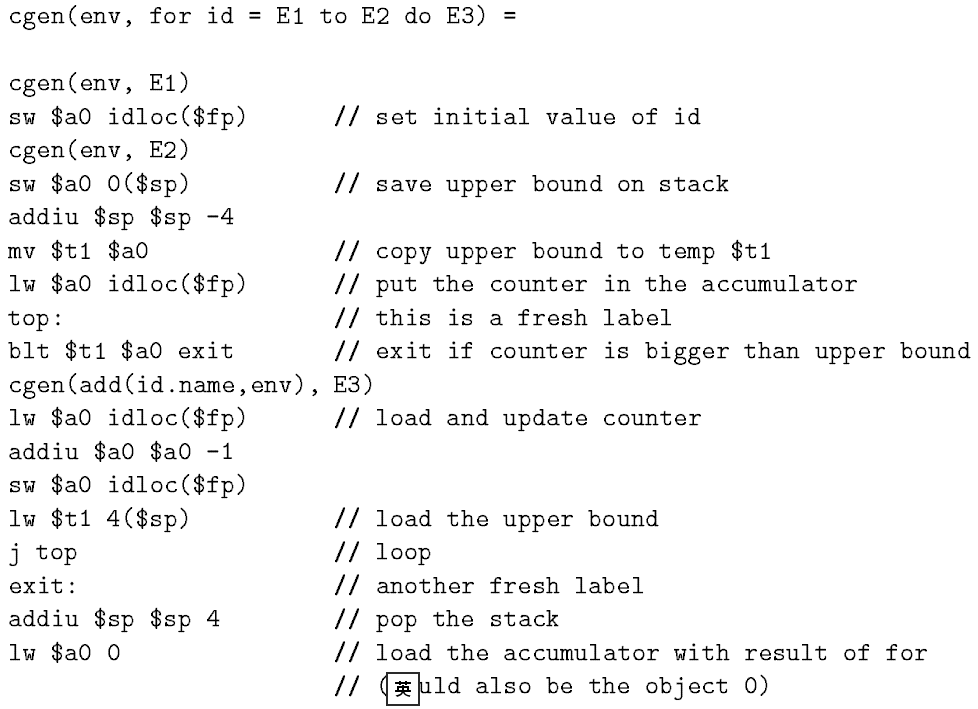
**Part (b).** The identifier introduced by a **for** expression is an iteration variable. A program should **not** explicitly update an iteration variables. We want to check whether a program updates some iteration variables or not using syntax-directed definitions. Write semantic rules for this task. Your semantic rules should use one inherited attribute *iset* and one synthesized attribute *legal*. *E.iset* is the set of iteration variables in scope of *E*, and *E.legal* is true if and only if no iteration variables are updated in *E*. Your solution should not use any other attributes.



**Part (c).** Write a code generation function *cgen* for **for** expression using stack machine scheme. You should show actual assembly code (similar pseudo-assembly is fine -- don't worry about MIPS syntax, please write comment for each statement if you use pseudo-assembly). Assume the following about the runtime environment:

* There is a frame pointer **$fp**, stack point **$sp**, accumlater **$a0**, and **$t1** for a temporary register
* The iteration variable *id* is stored as offset *idloc* in the frame.
* Use 32-bit integers (not integer objects).
* Temporaries are pushed on the stack, not stored in the frame.
* The net effect of evaluating *E* leaves the result of *E* in *$a0*, leaves *$sp* and *$fp* unchanged.

cgen(for id = E1 to E2 do E3) =



(env can be omitted)

**Question 2 (10+10= 20 Points). Garbage Collection**

**Part (a).** In Cool, suppose we want every object to support an additional method **hashValue** which returns an integer value such that

1. if we invoke **hashValue** multiple times on the same object, we always the get the same integer result;
2. for two distinct objects that are live simultaneously, there is a very small chance that they have the same **hashValue**. Recall that a live object is one that is reachable from the set of roots.

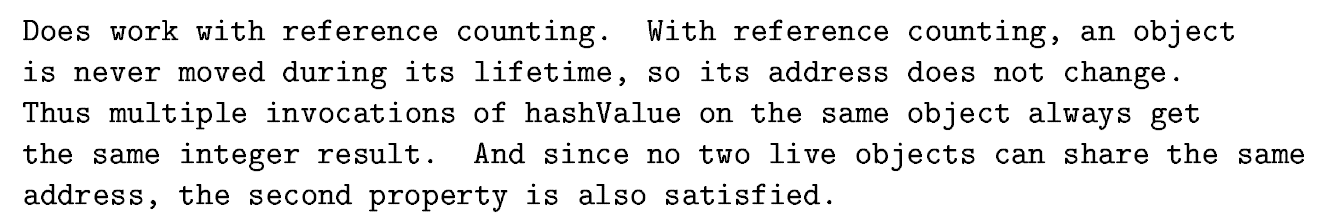
One implementation of **hashValue** is simply to return the address of the object invoking the **hashValue** method.

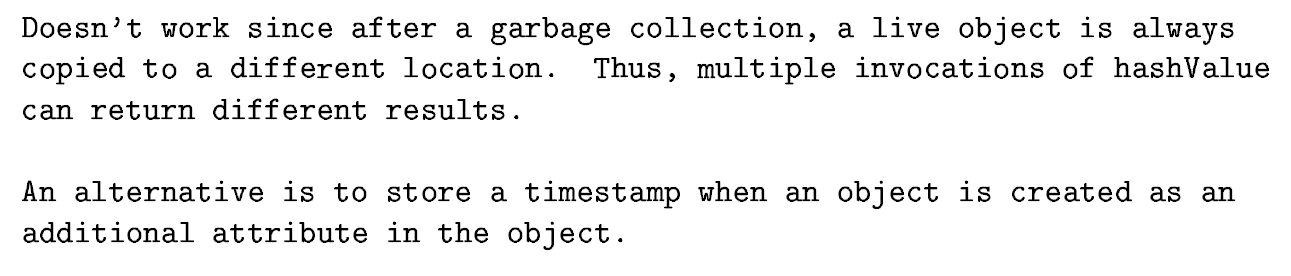
For each of the following garbage collection strategies, state whether or not the proposed implementation of **hashValue** works, briefly explain why or why not, and propose an alternative implementation for any cases where the proposed implementation does not work.

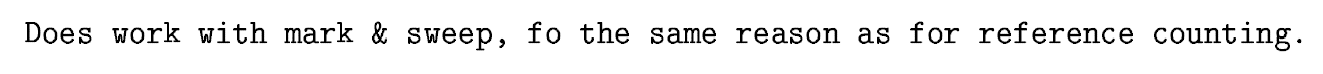
**1) reference counting**

**2) stop & copy**

**3) mark & sweep**

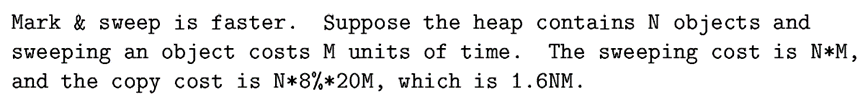
****

****

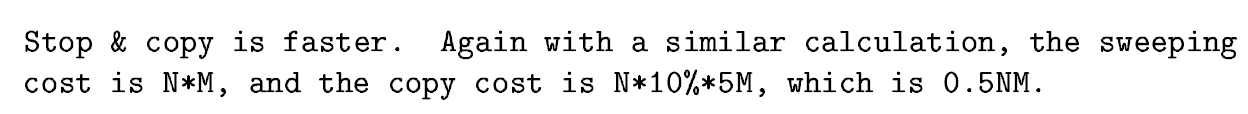
****

**Part (b).** Consider two garbage collection strategies: stop & copy, and mark & sweep, discussed in class. For each of the following situations, please answer which garbage collection strategy is faster than another one, and briefly justify your answer. Ignore the cost of allocation and of tracing in your answers, and focus on the cost of an actual garbage collection.

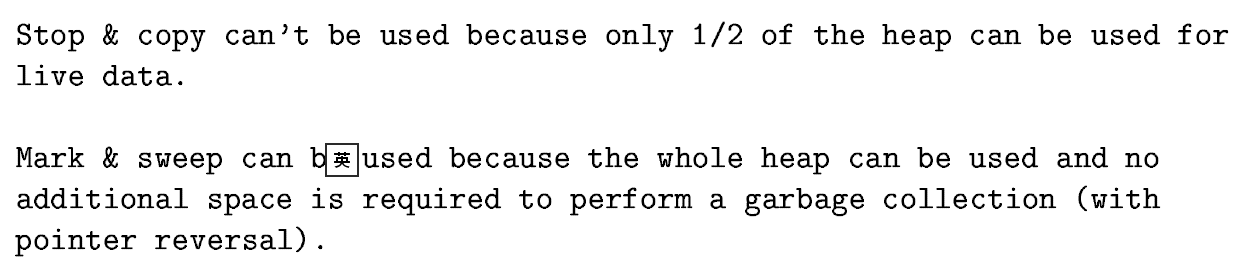
1. In the situation where there is a survival rate (percentage of live data after a collection) of 8%, and it is 20 times more expensive to copy an object than to sweep an object.

****

1. In the situation where the survival rate is 10% and it is five times more expensive to copy an object than to sweep an object.

****

1. In the situation where the size of live data is close to the size of the heap.

****

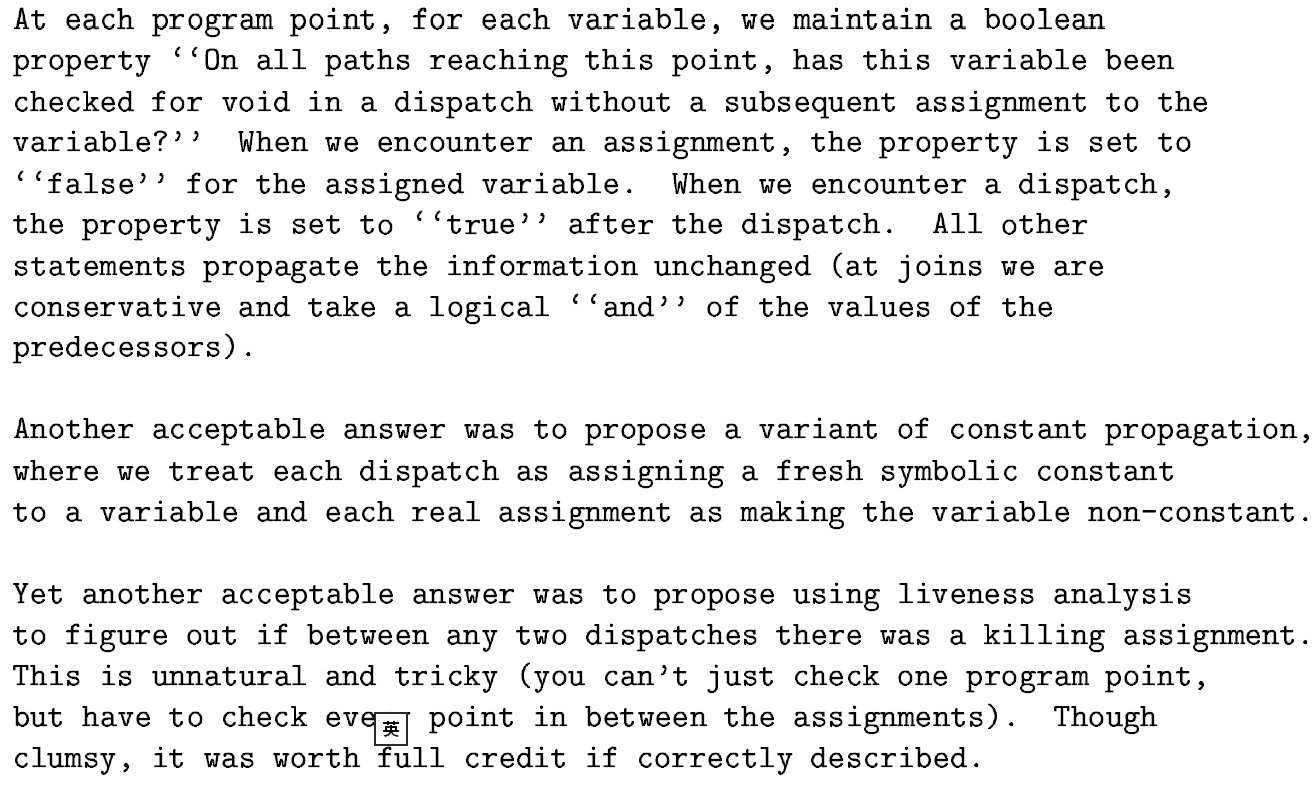
**Question 3 (5+5 =10 points).** Recall that one of Cool's runtime requirement is to detect dispatch to a void pointer. A naive implementation of this check requires a few additional instructions to be executed on every method call, which inevitably degrades the performance of Cool programs. In some situations, the dispatch on void check can be safely eliminated.

Consider the following excerpt from a Cool program,

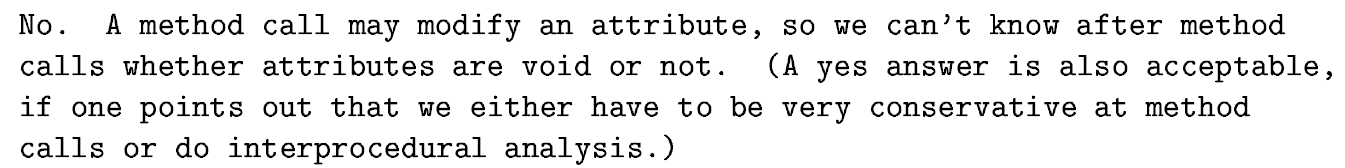
let x : A<- foo() in { x.bar(); // line 1 ...... x.baz() ; // line n}

We can eliminate the check for void in line n if it is known that the value of x has not changed since the last time we checked x for void in line 1. To check this condition, we need to perform a data flow analysis.

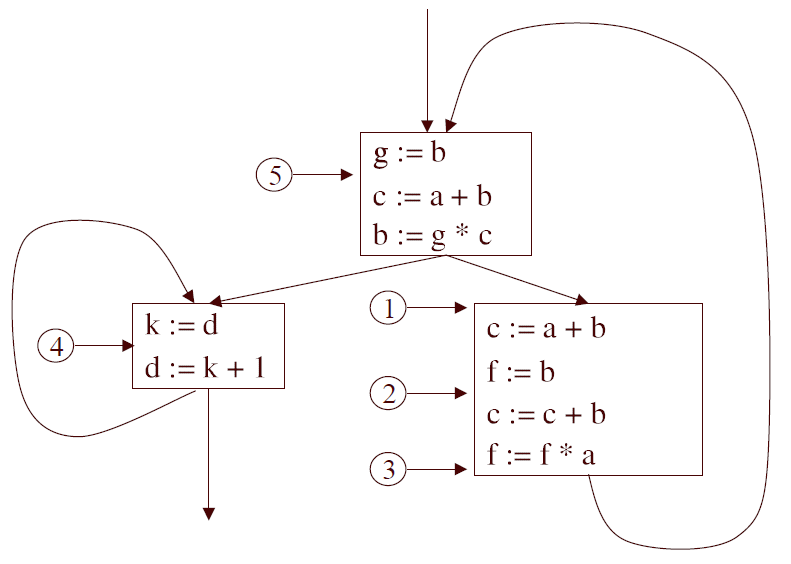
**Part (a).** Give an informal description what information does this analysis need to compute for each program point? How is that information used to decide whether tests for dispatch to void can be omitted?



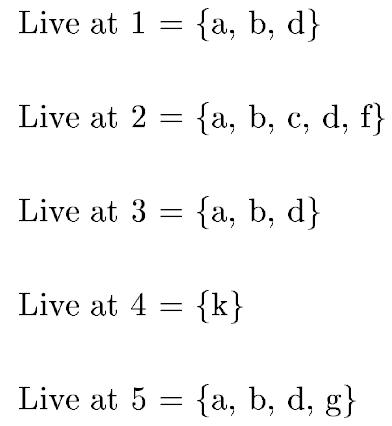
**Part (b).** Can we apply the method from above to eliminate the check if x is an attribute of an object rather than a variable? Why or why not?



**Question 4 (5+5+5=15points).** Consider the following program.

****

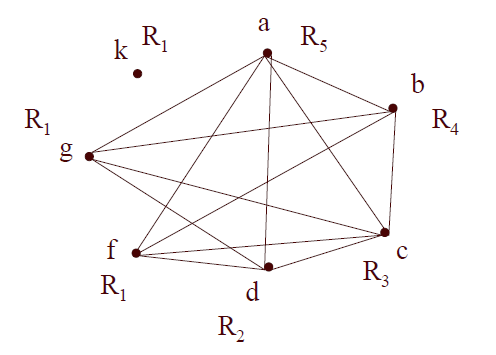
**Part (a).** Compute the set of live variables at the points 1-5 in the program. Assume that there are no live variables on exit.

****

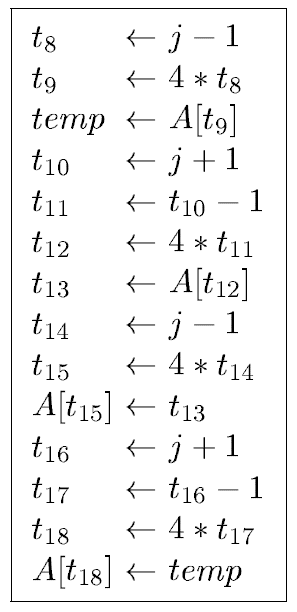
**Part (b).** Fill the edges in register interference graph of the program (you should use the same topology).



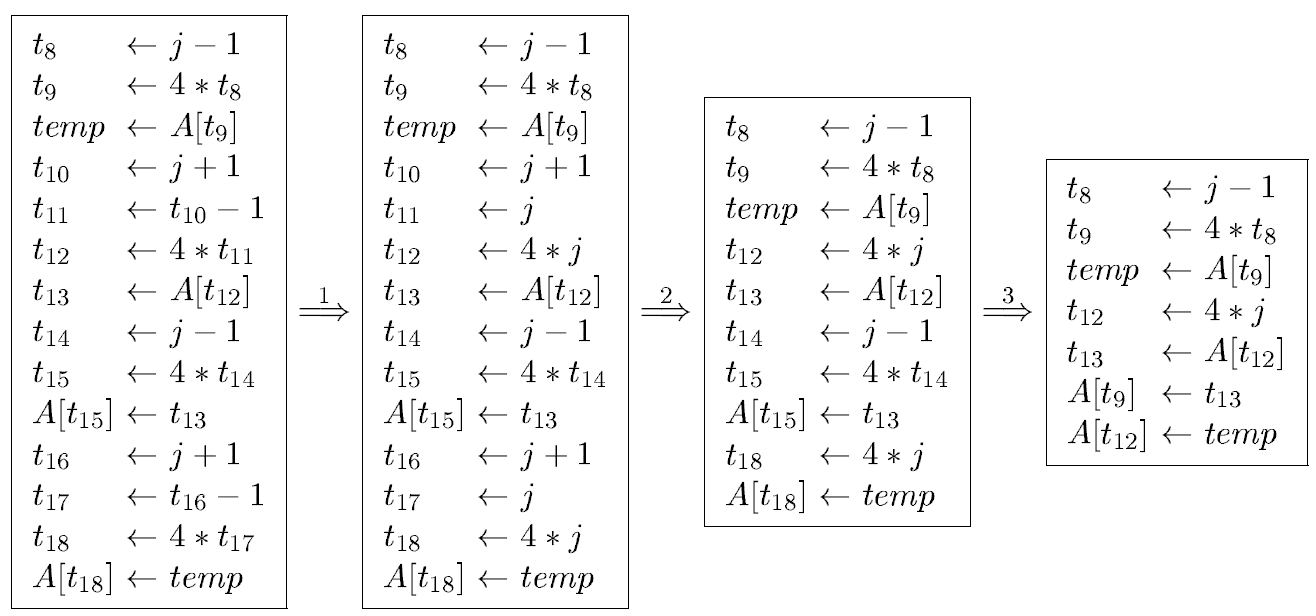
**Part (c).** Show a coloring for register interference graph using the minimal number of colors. Using color names such as R1, R2, R3,.... Write the color next to each node in your graph.

****

**Question 5 (10 points).** Consider the following basic block from a bubble-sort algorithm in three-address code.



The following is the evolution of the basic block through several optimization stages:



It may use the following optimizations:

* 1. Algebraic Simplification,
  2. Constant Folding,
  3. Copy propagation,
  4. Dead code elimination,
  5. Common subexpression elimination,
  6. Expression propagation,

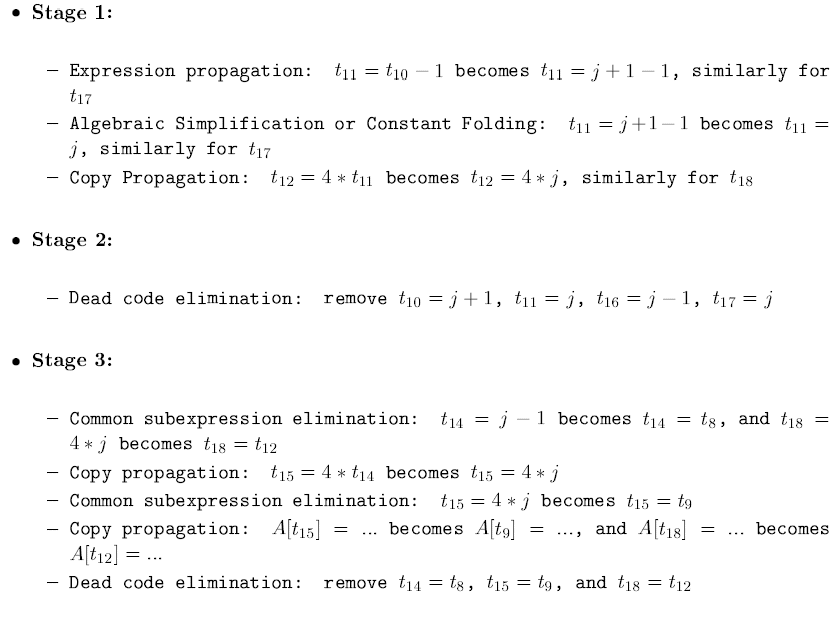
where Expression propagation is a new optimization which allow an entire expression in an assignment a<-e to be substituted for the later uses of a, assuming neither *a* nor any of the temporaries occurring in *e* are assigned to again before the use.

For each optimization stage, identify all the optimization(s) applied (there can be more than one in each stage). Write down the optimization names in no particular order of each stage.

**Stage 1:**

**Stage 2:**

**Stage 3:**

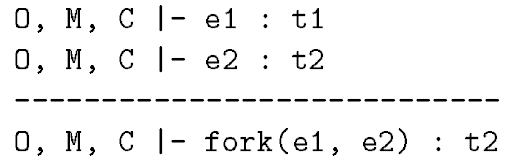
****

**Question 6 (5+5+5=10 Points).** We add a new kind of expression to extend Cool with concurrency:

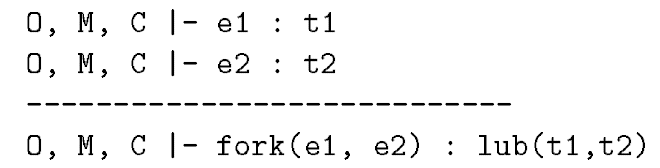
E🡪 … | **fork**(E,E)

The semantics of *fork(e1,e2)* is that the program may evaluate e1 and e2 in any order, including simultaneously. On a parallel machine, e1 and e2 could actually execute at the same time; on a sequential machine the two computations may be interleaved by partially evaluating e1, partially evaluating e2, switching back to e1, and so on.

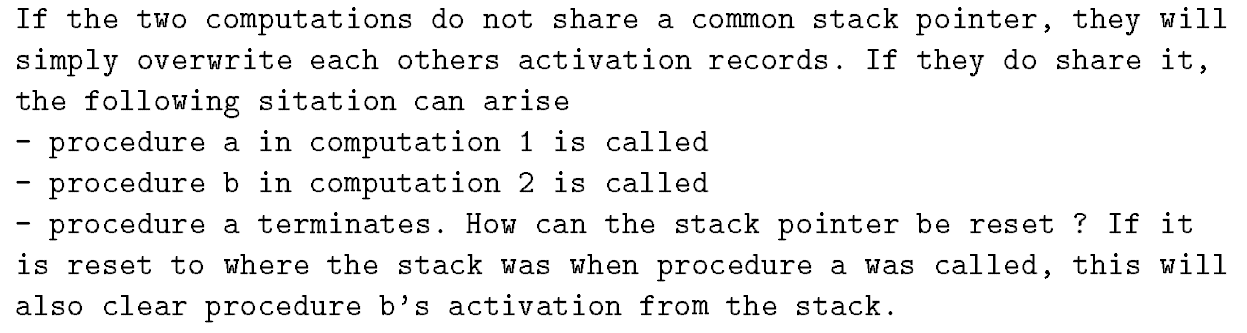
**Part (a)**. Assume the semantics of fork(e1,e2) is that both expressions are evaluated and the result is the value of e2. Write a Cool type rule for fork; make your rule as accurate as possible.



**Part (b)**. Assume the semantics of fork(e1,e2) is that both expressions are evaluated and the result is the value of the first expression that completes. Write a Cool type rule for fork; make your rule as accurate as possible.



**Part (c)**. Parallel evaluation of fork(e1,e2) implies that activation records cannot be stack allocated. Explain why this is the case. You are asked to explain the problem, instead of solve the problem.



**Question 7 (10 Points, extra credits).** In performing common subexpression elimination, we wish to find in the program two expressions that will always yield the same value, such that one can be replaced by the other. To do this, we need to compute available expressions.

An expression *x op y* is **available** at a point p of the program if

* all paths from the start of the program to point p must pass through the evaluation of *x op y*,
* and after the (last) evaluation of *x op y* there are no redefinitions of x or y.

This means that at point p we can replace *x op y* by the result of its last evaluation.

Problem statement: for each point in the program, determine if each expression is **available** in the program reaches the point.

Please formalize the available expression problem in the data flow analysis schema by giving the following definitions:

|  |  |
| --- | --- |
| Domain | All the expressions of the form b op c |
| Direction and equations | Forward  out[b] = fb(in[b])  in[b] =∩out[pred(b)] |
| Transfer function fb | fb(x) = (x-Killb)∪Genb |
| Meet Operation | ∩ |
| Boundary Condition | out[entry]=∅ |
| Initial interior points | out[b]={all expressions} |

For statement s (z := x op y),

Kills ={all the expressions containing z}

Gens={x op y}

For a basic block b,

fb is the composition of transfer functions of statements in the basic block b.

This means that

Killb is the set of expressions killed in b,

Genb is the set of expressions generated in b