CS243 Midterm Examination

Solutions

Winter 2015-2016

February 11th, 2016
3:00 pm - 4:15 pm

This exam is open book/laptop. We do not guarantee power though. You cannot access the Internet.

Duration: 75 minutes

Please do not post anything on Piazza until the solutions are put up on the class website.

Answer all questions on the exam paper itself.

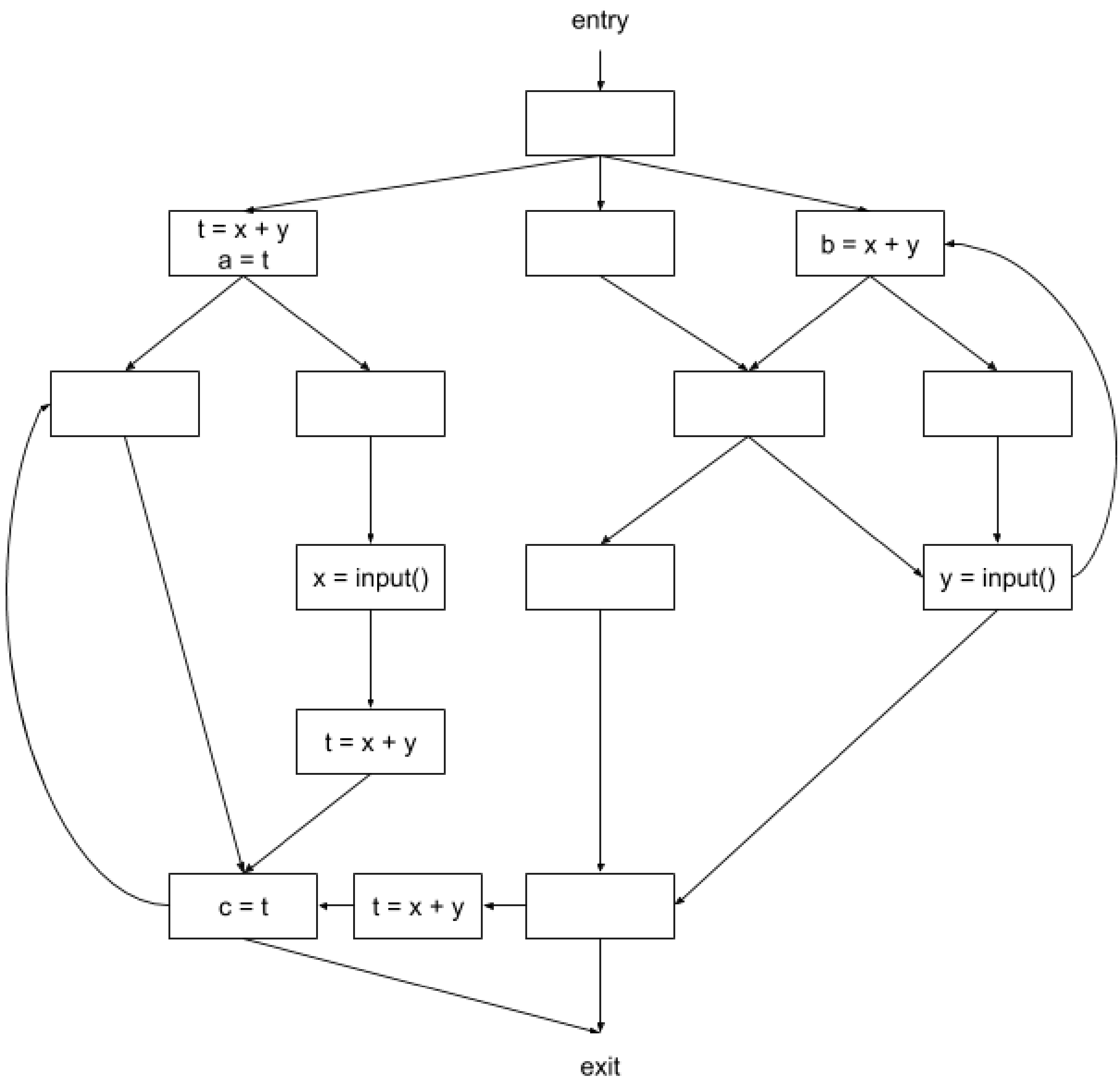
Write your name here: ______________________________

I acknowledge and accept the Stanford honor code.

(signed) ______________________________

Question	Marks	Score
1	14	14
2	12	12
3	12	12
4	22	22
Total	60	60

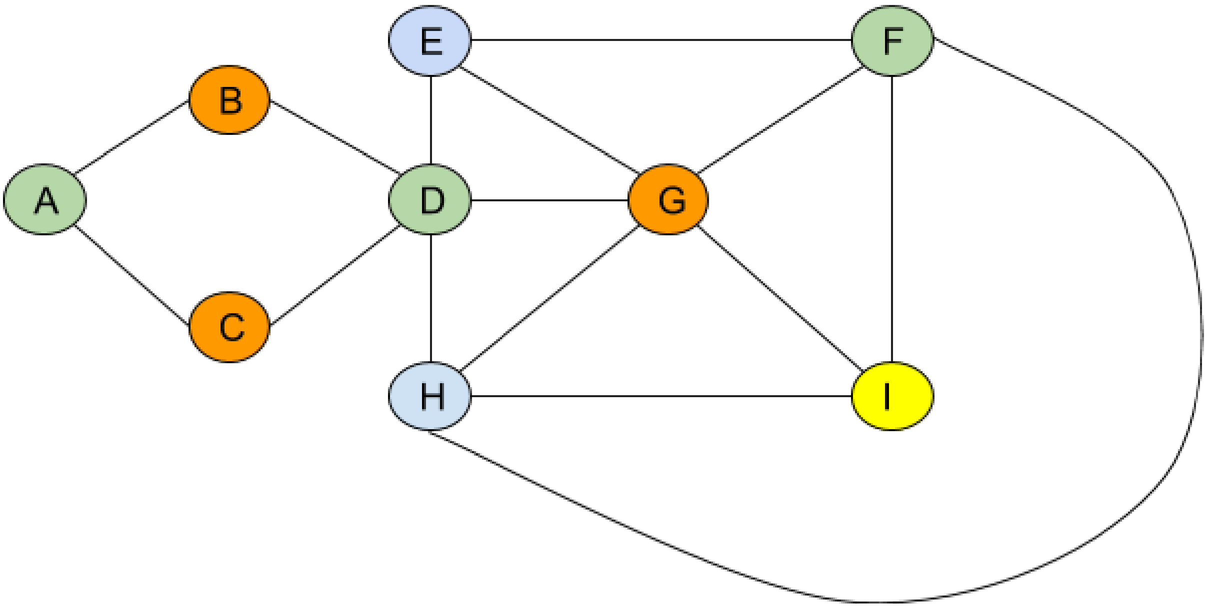
Q1) Partial Redundancy Elimination [14 marks]
Show the result of running PRE. You don't need to show any intermediate steps, only the final optimised flow graph.



2 pts for t=x+y, a=t
3 pts for x=input(), t=x+y
4 pts for t=x+y, c=t
5 pts for not touching b=x+y (Common mistake!)

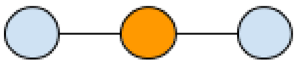
Q2) Register allocation [12 marks = 2 + 1 + 9]

Consider the following register interference graph:



a) True/False: In an interference graph, it is never possible to find a valid colouring for a node with degree = n, where n is the number of registers available. Provide a justification by way of a short proof or counterexample.

False. Counterexample with n = 2: (2 pts for a correct counter-example)



b) What is the minimum number of registers, R, required to avoid spilling for the above interference graph?

4 (1 pt for the number)

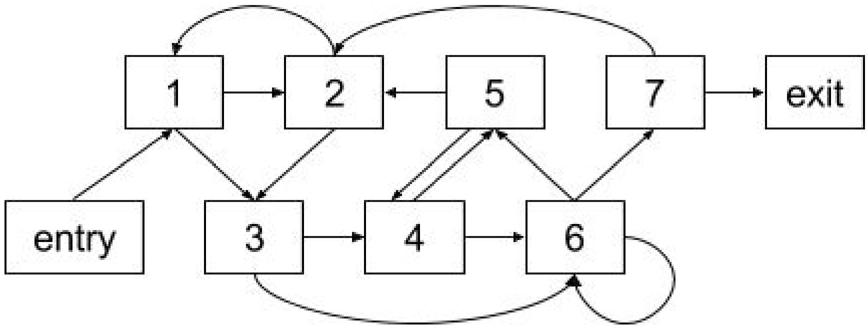
c) Show a register allocation that uses R number of registers for this graph. You can label the nodes in the graph with the register number assigned.

An example colouring is shown above.

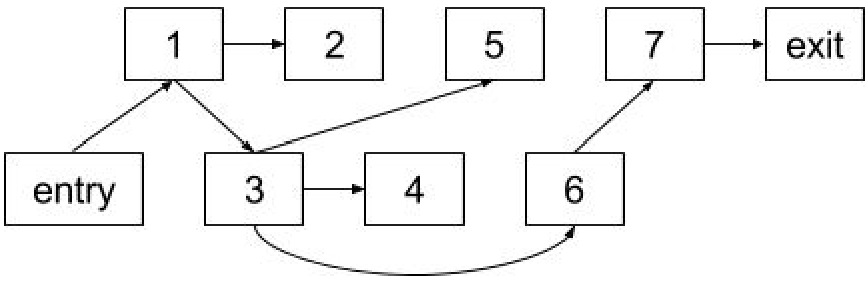
1 pt for each node correctly colored.

Q3) Dominators and Natural Loops [12 marks = 6 + 4 + 2]

Consider the following control flow graph:



a) Draw the dominator tree for the above graph



1 pt for each internal node (no pts for entry, exit and 1)

b) Identify all the back-edges and their corresponding natural loops

There are 2 back-edges:

1. 2->1: {1,2,3,4,5,6,7}
2. 6->6: {6}

1 pt for each of the above

c) Is the above graph reducible? (Yes/No) Provide the reason in a single line.

No. {4->5, 5->4, 2->3, 7->2, 5->2, 4->6, 6->5} are retreating edges that are not back-edges

2 pts for identifying at least 1 of the above edge. No pts for listing the definition of the reducibility

Q4) Simplified Pointer Analysis [22 marks = 2 + 10 + 10]

Pointers are an extremely useful method for indirect access but are also the cause of many bugs when not handled carefully. Improper allocation or deallocation of pointers can lead to dangling pointers and memory leaks which can cause segmentation faults in run time.

Consider a simplified programming language with two kinds of statically typed variables: (1) integers, or (2) pointers to integers. For simplicity, we use p, q to represent pointer variables, and u, v, w to represent integer variables. It has the following statements:

ASSIGNMENT:	v = <constant>
COPY INT VARIABLES:	u = v
ADDITION:	w = u + v
COND BRANCH:	if (v) goto L
ALLOCATION:	p = new()
LOAD INT INDIRECTLY:	v = *p
STORE INT INDIRECTLY:	*p = v
DELETION:	delete(p)

Your task is to develop a static program analysis that will issue the following warnings:

- Type 1 Warning (Dangling pointer): Issue a warning on any statement that may dereference or delete an invalid pointer. A pointer is invalid if has not been allocated, or if it has been freed.
- Type II Warning (Memory Leak). Issue a warning on an allocation statement if the pointer MAY not have been deleted before the program exits.

For example for the following code:

```
L1: p = new()
L2: v = 2
L3: *p = v
L4: if(v) goto L6
L5: delete(p)
L6: v = *p
L7: if(v) goto L9
L8: p = new()
L9: delete(p)
```

a. Warning I should be issued for: {L6, L9}
b. Warning II should be issued for: {L1}

You can design **one or more** data flow analyses to answer the two parts of this problem. On the next page is a template that you need to fill out for each analysis.

State explicitly how you generate the warnings and the causes of the warnings.

Warning I is generated at deletion and load/store int indirectly statements if the pointer is not in IN[B]

Warning II is generated at the allocation statement if the pointer is in OUT[B]

1 pt each for the explanation above
For the table:

- **Direction - 1pt**
- **Semi-lattice domain - 0.5 pt**
- **Semi-lattice diagram - 2 pts**
- **Meet operator - 1pt**
- **Transfer function - 3 pts**
- **Boundary condition - 0.5 pt**
- **Interior point - 0.5 pt**
- **Monotone - 0.5 pt**
- **Distributive - 0.5 pt**
- **Convergence - 0.5 pt**

Common Mistakes:

- **Not treating a statement as a basic block and running into Gen, Kill precedence issues**
- **Not specifying IN or OUT when generating warnings**
- **Semi-lattice diagram not complete or interchanged top and bottom**

Data Flow Analysis Template (Warning I)	
Property	**Answer**
Direction of Analysis (Forward/Backward)	**Forward**
Meaning of the values in the semi-lattice	**Set of allocated pointers**
Semi-lattice diagram (Mark the top and bottom clearly)	**Top = U (universal) → subsets decreasing one at a time → {} = Bot**
Meet Operator	∩
Transfer function of a basic block	**For allocation: OUT[s] = IN[s] + {p}** **For deletion: OUT[s] = IN[s] - {p}** **For the rest: OUT[s] = IN[s]** **Transfer function for basic block can be composed from the above transfer function for a statement**
Boundary condition (assignment to OUT[entry]/IN[exit])	**OUT[entry] = {}**
Interior Points (assignment to IN[B]/OUT[B])	**OUT[B] = U**
Is the framework monotone? (Yes/no: No explanation needed)	**Yes**
Is the framework distributive? (Yes/no: No explanation needed)	**Yes**
Will the algorithm converge? (Yes/no: No explanation needed)	**Yes**

Data Flow Analysis Template (Warning II)	
Property	**Answer**
Direction of Analysis (Forward/Backward)	**Backward**
Meaning of the values in the semi-lattice	**Set of escaped pointers**
Semi-lattice diagram (Mark the top and bottom clearly)	**Top = {} → subsets increasing one at a time → U = Bot**
Meet Operator	**U**
Transfer function of a basic block	**For deletion: IN[s] = OUT[s] - {p}** **For allocation: IN[s] = OUT[s] + {p}** **For the rest: IN[s] = OUT[s]** **Transfer function for basic block can be composed from the above transfer function for a statement**
Boundary condition (assignment to OUT[entry]/IN[exit])	**IN[exit] = U**
Interior Points (assignment to IN[B]/OUT[B])	**IN[B] = {}**
Is the framework monotone? (Yes/no: No explanation needed)	**Yes**
Is the framework distributive? (Yes/no: No explanation needed)	**Yes**
Will the algorithm converge? (Yes/no: No explanation needed)	**Yes**