Name:

Midterm 2

CMSC 430
Introduction to Compilers
Fall 2013

November 20, 2013

Instructions

This exam contains 9 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		20
2		25
3		20
4		20
5		15
Total		100

Question 1. Short Answer (20 points).

a. (5 points) In class we've discussed three virtual machines: the Lua VM, the Java VM, and the Dalvik VM. List two differences that you can observe among the VMs, e.g., something different between Lua VM and the Java VM, etc.

b. (5 points) Is dataflow analysis guaranteed to compute the *meet over all paths* (MOP) solution? Explain your answer.

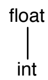
c. (5 points) What are the three possible representations of two-dimensional arrays that we discussed in class? Explain the differences between them. Feel free to draw pictures if it's helpful.

d. (5 points) What is a *virtual method table*?

Question 2. Types (25 points).

a. (7 points) Suppose *int* is a subtype of *float*, and consider the following four types:

$$int \rightarrow int \qquad int \rightarrow float \qquad float \rightarrow int \qquad float \rightarrow float$$

Draw a partial order diagram showing the $\leq$ relationships between these four types implied by the standard subtyping rules. For example, if we asked you to draw the relationship between *int* and *float*, you would draw the following: 

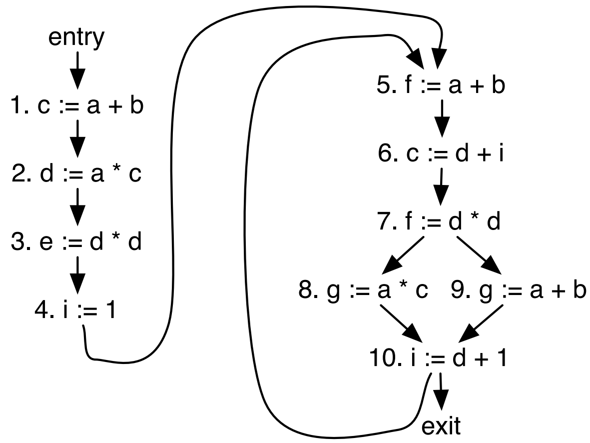
Also indicate which element is $\top$, and which element is $\bot$, if any.

b. (8 points) Write down every type t such that $((int \rightarrow float) \rightarrow float) \leq t$, following standard subtyping rules.

c. (10 points) Fill in the following table with either an *untyped* (i.e., no type parameter annotations) lambda calculus term (on the left) or its corresponding type according to the type inference algorithm we saw in class (on the right). We've filled in the first row as an example. Remember the scope of λ extends as far to the right as possible. For example, $\lambda x.\lambda y.x\ y$ is parsed as $\lambda x.\lambda y.(x\ y)$.

Term	Type
$\lambda x.x$	$\alpha \rightarrow \alpha$
$\lambda x.3$	
	$\alpha \rightarrow \beta \rightarrow \alpha$
$\lambda x.\lambda y.x\ y$	
	$(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$
$\lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$	
$\lambda x.x\ (\lambda y.3)$	
	$(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

Question 3. Data flow analysis (20 points). Consider the following control-flow graph.



a. (10 points) Write down the sets of **live variables** at the *beginning* of each statement.

Statement	Variables live at beginning of statement
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

b. (10 points) Write down the set of **available expressions** at the *end* of each statement.

Statement	Expressions available at end of statement
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Question 4. Data flow analysis design (20 points). The goal of *sign analysis* is to determine, for each variable in the program, whether it is positive, negative, or zero. In this problem, you will design a data flow analysis that implements sign analysis.

a. (5 points) Should the analysis be forward or backward?

b. (5 points) What should be the lattice for sign analysis? Draw a picture to help your explanation. *Note:* You do not need to worry about various combinations of signs (e.g., "positive or zero").

c. (5 points) Suppose $x = +$, $y = -$, and $z = 0$ just prior to each for the following statements. Write down the dataflow fact about a just after the statement.

$a := x + x$	
$a := x * y$	
$a = x - y$	
$a := x + y$	

d. (5 points) If we implement the usual dataflow analysis algorithm, is the algorithm guaranteed to terminate? Why or why not?

Question 5. Code generation (15 points). Below is a snippet of `08-codegen-2.ml` from class, showing the input expression language, the "bytecode" instruction language, and compilation. To save some writing, we renamed `‘L_Register` to `‘L_Reg`.

```
type expr =
  | EInt of int
  | EAdd of expr * expr
  | ESub of expr * expr
  | EMul of expr * expr
  | EId of string
  | EAssn of string * expr
  | ESeq of expr * expr
  | EIfZero of expr * expr * expr

type reg = [ ‘L_Reg of int ]
type src = [ ‘L_Int of int | ‘L_Ptr of int ]
type dst = [ ‘L_Ptr of int ]

type instr =
  | ILoad of reg * src         (* dst, src *)
  | IStore of dst * reg        (* dst, src *)
  | IAdd of reg * reg * reg (* dst, src1, src2 *)
  | IMul of reg * reg * reg (* dst, src1, src2 *)
  | IIfZero of reg * int       (* guard, target *)
```

```
  | IJmp of int                (* target *)
  | IMov of reg * reg          (* dst, src *)

let rec comp_expr (st:(string*int) list) = function
  | EInt n ->
      let r = next_reg () in
        (r, [ILoad (‘L_Reg r, ‘L_Int n)])
  | EIfZero (e1, e2, e3) ->
      let (r1, p1) = comp_expr st e1 in
      let (r2, p2) = comp_expr st e2 in
      let (r3, p3) = comp_expr st e3 in
      let r = next_reg () in
        (r, p1 @
           [IIfZero (‘L_Reg r1, (2+(List.length p3)))] @
           p3 @
           [IMov (‘L_Reg r, ‘L_Reg r3);
            IJmp (1+(List.length p2))] @
           p2 @
           [IMov (‘L_Reg r, ‘L_Reg r2)]
        )
```

Suppose we extend the source language with a *for loop* `EFor(e1, e2, e3, e4)` roughly corresponding to the C construct `for (e1; e2; e3) e4`. Here `e1` is the initialization, `e2` is the guard, `e3` is the increment, and `e4` is the loop body. The loop body should be executed if the `guard` is **non-zero**. The whole construct should return 0 as a result.

Write a case of comp_expr that compiles `EFor`.

```
let rec comp_expr (st:(string*int) list) = function
  ...
  | EFor (e1, e2, e3, e4) ->
```