Name:

Midterm 2

CMSC 430
Introduction to Compilers
Fall 2014

November 19, 2014

Instructions

This exam contains 10 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		25
2		25
3		10
4		25
5		15
Total		100

Question 1. Short Answer (25 points).

a. (5 points) Briefly explain what a *virtual method table* (or *vtable*) is and what it's used for.

b. (5 points) List three potential goals of *optimization* in a compiler.

c. (5 points) Briefly explain what the *progress theorem* is.

d. (5 points) What do *mutation* and *crossover* have to do with automated program repair, as discussed in class? Explain very briefly.

e. (5 points) Briefly explain what an *activation record* is and list 3 items in an activation record.

Question 2. Type Systems (25 points).

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

b. (2 points) Assume that $int < float$. Write down every type t such that $t \leq int\ ref \rightarrow float\ ref$, 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).

Term	Type
	int
$\lambda x.x$	
	$\alpha \to \beta \to \beta$
$\lambda x.\lambda y.y\ x$	
$\lambda x.x\ 3$	

d. (5 points) Recall the simply typed lambda calculus:

$$\begin{array}{rcl} e & ::= & n \mid x \mid \lambda x{:}\,t.e \mid e\ e \\ t & ::= & int \mid t \to t \\ A & ::= & \emptyset \mid x{:}\,t, A \end{array}$$

$$\textsc{Int}\ \frac{}{A \vdash n : int} \qquad \textsc{Var}\ \frac{}{A \vdash x : A(x)} \qquad \textsc{Lam}\ \frac{x{:}\,t, A \vdash e : t'}{A \vdash (\lambda x{:}\,t.e) : t \to t'} \qquad \textsc{App}\ \frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\ e_2 : t'}$$

Draw a derivation that the following type judgment holds, where $A = +{:}\,int \to int \to int$. (You can draw the derivation upward from the judgment, and you can write i instead of int to save time):

$$A \vdash (\lambda x{:}\,int. +\ x)\ 1 : int \to int$$

Question 3. Interpreter Implementation (10 points). Below is a snippet of the bytecode interpreter code from `06-codegen-2.ml`.

```
type src = [ `Const of int | `Ptr of int ]
type reg = [ `Reg of int ]
type dst = [ `Ptr of int ]

type symtbl = (string * int) list
type heap = (int, int) Hashtbl.t
type regs = (int, int) Hashtbl.t

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 run_instr (h:heap) (rs:regs) = function
  | IAdd (`Reg r1, `Reg r2, `Reg r3) ->
     Hashtbl.replace rs r1 ((Hashtbl.find rs r2) +
                            (Hashtbl.find rs r3)); None
  | IIfZero (`Reg r, n) ->
     if (Hashtbl.find rs r) = 0 then Some n else None
  | ...
```

Suppose we make the unfortunate decision to modify our bytecode language to have a special undefined value, like JavaScript. We begin by introducing a new type, intOrUndef, to stand for either the undefined value or an integer; we add a new instruction, IIfUndef (r, n); and we adjust the types src, heap, and regs appropriately:

```
type intOrUndef = Undef | Int of int

type instr = ... | IIfUndef of reg * int

type src = [ `Const of intOrUndef | `Ptr of int ]
type heap = (int, intOrUndef) Hashtbl.t
type regs = (int, intOrUndef) Hashtbl.t
```

The desired semantics is as follows:

- IIfUndef (r,n) branches by n if r contains Undef, otherwise it falls through.
- If Undef if used as either argument to addition, the result should be Undef.
- If Undef is used as the guard of IIfZero, it should be treated as false (i.e., as a non-zero value).

Rewrite the cases in run_instr for IIfUndef, IAdd, and IIfZero to implement this semantics. You can write a helper function if you want. You do not need to implement any other parts of run_instr.

```
let rec run_instr (h:heap) (rs:regs) = function
```

Question 4. Code Generation (25 points). Below is more code from `06-codegen-2.ml`, showing the input expression and part of the compiler.

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

type symtbl = (string * int) list

let rec comp_expr (st:symtbl) = function
  | EInt n ->
      let r = next_reg () in
        (r, [ILoad (`Reg r, `Const n)])
  | EPlus (e1, e2) ->
      let (r1, p1) = comp_expr st e1 in
      let (r2, p2) = comp_expr st e2 in
      let r = next_reg () in
        (r, p1 @ p2 @ [IAdd (`Reg r, `Reg r1, `Reg r2)])
  | 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 (`Reg r1, (2 + (List.length p3)))] @
            p3 @
            [IMov (`Reg r, `Reg r3);
             IJmp (1 + (List.length p2))] @
            p2 @
            [IMov (`Reg r, `Reg r2)]
        )
```

a. (10 points) Suppose we extend the source language with a *repeat-until loop* ERepeat(e1, e2), meaning "repeat e1 until e2 becomes non-zero." Note that a repeat-until loop always executes the body e1 at least once (so it evaluates e1; checks if e2 is non-zero; if not evaluates e1 again; etc). Write a case of comp_expr that compiles ERepeat. The loop itself should evaluate to 0.

```
let rec comp_expr (st:symtbl) = function
  ...
  | ERepeat (e1, e2) ->
```

b. (15 points) Now consider again adding an undefined value to the language:

```
type expr =
 | EUndef
 | ...
```

Write the EUndef case of comp_expr. Also, rewrite the EPlus case of comp_expr to implement the Undef semantics *without* relying on the special IAdd handling that understands Undef. That is, your compiled output code should *only* call IAdd with integer arguments.

```
let rec comp_expr (st:symtbl) = function
  ...
  | EUndef →
```

```
  ...
  | EPlus (e1, e2) →
```

Question 5. Optimization (15 points). In each row of the table, perform the indicated optimization (and *only* that optimization), writing down the optimized code on the right-hand side of the table. To reduce writing, we write rn instead of 'Reg n, and we write n instead of 'Const n.

Initial code	Optimized code
ILoad (r0, 42) IMov (r1, r0) IAdd (r2, r0, r1)	Copy propagation
IAdd (r3, r1, r2) IMul (r4, r1, r2) IAdd (r5, r1, r2)	Common subexpression elimination
ILoad (r0, 42) ILoad (r1, 3) IAdd (r3, r0, r1) IMul (r4, r1, r2) IAdd (r5, r1, r3)	Constant folding
ILoad (r0, 0) ILoad (r1, 1) ILoad (r2, 2) IAdd (r3, r0, r1) IAdd (r4, r0, r2) IAdd (r5, r1, r3) *(∗ assume only r5 is live ∗)*	Dead code elimination