

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.

Answer: It is a collection of methods for a particular class. Each instance of a class has a pointer to the virtual method table for that class. When one of an object's instance methods is invoked, it is resolved by looking it up in the virtual method table.

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

Answer: Improve running time, decrease memory usage, reduce code size. Other reasonable answers were also acceptable.

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

Answer: A well-typed term is either a value, or it can be reduced a step.

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

Answer: The automated program repair algorithm presented used mutation (changing a statement in the program) and crossover (combining together two potential repaired programs) to generate candidate repaired programs.

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

Answer: An activation record is a stack frame that contains information about a function invocation, including the parameters, return address, local variables, space to temporarily save registers, and possibly the caller's activation record pointer.

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.

Answer:

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

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.

Answer:

$int\ ref \rightarrow float\ ref$

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
3	int
$\lambda x.x$	$\alpha \rightarrow \alpha$
$\lambda x.\lambda y.y$	$\alpha \rightarrow \beta \rightarrow \beta$
$\lambda x.\lambda y.y x$	$\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$
$\lambda x.x 3$	$(int \rightarrow \alpha) \rightarrow \alpha$

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

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

$$\begin{array}{c}
 \text{INT} \\
 \hline
 A \vdash n : int \\
 \\
 \text{VAR} \\
 \hline
 A \vdash x : A(x) \\
 \\
 \text{LAM} \\
 \frac{x:t, A \vdash e : t'}{A \vdash (\lambda x:t.e) : t \rightarrow t'} \\
 \\
 \text{APP} \\
 \frac{A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t}{A \vdash e_1 e_2 : t'}
 \end{array}$$

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

Answer:

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

$$A \vdash (\lambda x: int. + x) 1 : int \rightarrow int$$

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

<pre> 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 *) </pre>	<pre> IMul of reg * reg * reg (* dst, src1, src2 *) IIfZero of reg * int (* guard, target *) IJump 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 ... </pre>
---	--

Suppose we make the unfortunate decision to modify our bytecode language to have a special `undef` 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:

<pre> type intOrUndef = Undef Int of int type instr = ... IIfUndef of reg * int </pre>	<pre> type src = ['Const of intOrUndef 'Ptr of int] type heap = (int, intOrUndef) Hashtbl.t type regs = (int, intOrUndef) Hashtbl.t </pre>
--	---

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**

Answer:

```

  | IAdd ('Reg r1, 'Reg r2, 'Reg r3) →
      Hashtbl.replace rs r1 (stupid_add (Hashtbl.find rs r2),
                                       (Hashtbl.find rs r3)); None
  | IIfZero ('Reg r, n) →
      match (Hashtbl.find rs r) with
      | Int 0 → Some n
      | _ → None
  | IIfUndef ('Reg r, n) →
      match (Hashtbl.find rs r) with
      | Undef → Some n
      | _ → None
  | ...

let stupid_add = function
  | Int x, Int y → x + y
  | _ → Undef

```

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

<pre> type expr = EInt of int EPlus of expr * expr EMul of expr * expr Eld 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 </pre>	<pre> 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)]) </pre>
---	---

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) →

```

Answer:

```

let (r1, p1) = comp_expr st e1 in
let (r2, p2) = comp_expr st e2 in
let r = next_reg () in
(r, p1 @
  p2 @
  [IIfZero ('L_Register r2, -((List.length p1) + (List.length p2) + 1));
   ILoad ('L_Register r, 'L_Int 0)])

```


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 →
```

Answer:

```
    let r = next_reg () in  
    (r, [ILoad(r, 'Const UnDef)])
```

```
  ...  
  | EPlus (e1, e2) →
```

Answer:

```
    let (r1, p1) = comp_expr st e1 in  
    let (r2, p2) = comp_expr st e2 in  
    let r = next_reg () in  
    (r, p1 @  
      p2 @  
      [ILfUnDef(r1, 3);  
       ILfUnDef(r2, 2);  
       IAdd(r, r1, r2);  
       IJump 1  
       ILoad(r, 'Const UnDef)])
```

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 *m* instead of 'Reg *n*', and we write *n* instead of 'Const *n*'.

Initial code	Optimized code
<pre>ILoad (r0, 42) IMov (r1, r0) IAdd (r2, r0, r1)</pre>	<p>Copy propagation</p> <p>Answer:</p> <pre>ILoad (r0, 42) IMov (r1, r0) IAdd (r2, r0, r0)</pre>
<pre>IAdd (r3, r1, r2) IMul (r4, r1, r2) IAdd (r5, r1, r2)</pre>	<p>Common subexpression elimination</p> <p>Answer:</p> <pre>IAdd (r3, r1, r2) IMul (r4, r1, r2) IMov (r5, r3)</pre>
<pre>ILoad (r0, 42) ILoad (r1, 3) IAdd (r3, r0, r1) IMul (r4, r1, r2) IAdd (r5, r1, r3)</pre>	<p>Constant folding</p> <p>Answer:</p> <pre>ILoad (r0, 42) ILoad (r1, 3) ILoad (r3, 45) IMul (r4, r1, r2) ILoad (r5, 48)</pre>
<pre>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 *)</pre>	<p>Dead code elimination</p> <p>Answer:</p> <pre>ILoad (r0, 0) ILoad (r1, 1) IAdd (r3, r0, r1) IAdd (r5, r1, r3)</pre>