

Name:

Midterm 2

CMSC 430
Introduction to Compilers
Fall 2015

November 11, 2015

Instructions

This exam contains 8 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		30
3		20
4		30
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Briefly describe what a *basic block* is.

Answer: A basic block is a sequence of statements (usually in three-address code) such that there are no jumps into the block except for the first statement, and no jumps out of the block except for the last statement.

b. (5 points) Are the kinds of optimizations discussed in class guaranteed to make programs go faster? Explain briefly.

Answer: No, optimizations are heuristic, and while they aim to make programs faster, they could potentially make no difference or even make programs go slower. A major reason is that perturbations in the program text could interfere with caching or other performance optimizations in the hardware.

c. (5 points) List three distinct items that might be part of a function's *activation record* or *stack frame*.

Answer: Parameters, saved registers, the return value, the return address, local variables.

d. (5 points) Briefly explain the relationship between an interpreter and a symbolic executor.

Answer: An interpreter steps through a program, executing its semantics using concrete values. A symbolic executor is similar, except certain inputs can be marked as symbolic. When symbolic values are used in branches, a symbolic executor may fork execution to explore multiple paths.

Question 2. Code Generation (30 points). Below is code from 06-codegen-2.ml, showing the input expression language and part of the compiler. We've eliminated global variables since they're not needed for this problem.

<pre> type expr = Elnt of int EAdd of expr * expr EMul of expr * expr ESeq of expr * expr ElfZero of expr * expr * expr type reg = ['Reg of int] type src = ['Int of int] type instr = ILoad of reg * src (* dst, src *) IAdd of reg * reg * reg (* dst, src1, src2 *) IMul of reg * reg * reg (* dst, src1, src2 *) IlfZero of reg * int (* guard, target *) IJmp of int (* target *) IMov of reg * reg (* dst, src *) </pre>	<pre> let rec comp_expr = function Elnt n → let r = next_reg () in (r, [ILoad ('Reg r, 'Int n)]) ... ElfZero (e1, e2, e3) → let (r1, p1) = comp_expr e1 in let (r2, p2) = comp_expr e2 in let (r3, p3) = comp_expr e3 in let r = next_reg () in (r, p1 @ [IlfZero ('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 add a construct `EDoWhile(e1, e2)` that executes `e1` while `e2` in non-zero and terminates, returning 0, when `e2` becomes 0. (Note this means `e1` will always be executed at least once.) Add a case to `comp_expr` for compiling `EDoWhile`.

```

let rec comp_expr = function
  ...
  | EDoWhile (e1, e2) →

```

Answer:

```

let (r1, p1) = comp_expr e1 in
let (r2, p2) = comp_expr e2 in
let r = next_reg () in
(r, p1 @
  p2 @
  [IlfZero ('Reg r2, 1);
   IJmp (-(List.length p1) + (List.length p2) + 2));
  ILoad ('Reg r, 'Int 0)])

```

b. (5 points) Suppose we extend the language with *local variables*. We add `ELocRd(x)` to indicate a read of local variable `x`, where `x` is a string, and `ELocWr(x, e)` to indicate writing the value of expression `e` to local variable `x`.

Fill in the missing code of the following function `vars_of : expr -> string list` so it returns a list of the variables used in an expression. Assume you have a function `uniq : string list -> string list` that returns its input list with duplicates removed.

```
let rec vars_of' = function

| EInt _ -> []

| EAdd (e1, e2) | ESub (e1, e2) | EMul (e1, e2)

| ESeq (e1, e2) | EDoWhile (e1, e2) -> (vars_of e1) @ (vars_of e2)

| ELocRd x -> [x]

| ELocWr (x, e) -> x :: (vars_of e)

| EIfZero (e1, e2, e3) -> (vars_of e1) @ (vars_of e2) @ (vars_of e3)

let vars_of e = uniq (vars_of' e)
```

c. (15 points) Using `vars_of`, write a new function `comp : expr -> instr list` that builds a symbol table for the expression and then calls `comp_expr` to compile it using that symbol table. (Note this may be a slightly different approach for locals than you took in Project 4.) Also modify the signature of `comp_expr` appropriately and write down the cases for `ELocRd` and `ELocWr` in `comp_expr`. (You don't need to modify the other cases of `comp_expr` to match the new signature.)

Answer:

```
let rec comp_expr symtab = function
...
| ELocRd x ->
  let r = next_reg () in
  (r, [IMov ('Reg r, 'Reg (List.assoc x symtab))])
(* OR *)
| ELocRd x ->
  ((List.assoc x symtab), [])
| ELocWr (x, e) ->
  let (r, p) = comp_expr st e in
  (r, p @ [IMov ('Reg (List.assoc x symtab), 'Reg r)])

let rec mksymtab = function
[] -> []
| (x :: xs) -> (x, next_reg ()) :: (mksymtab xs)

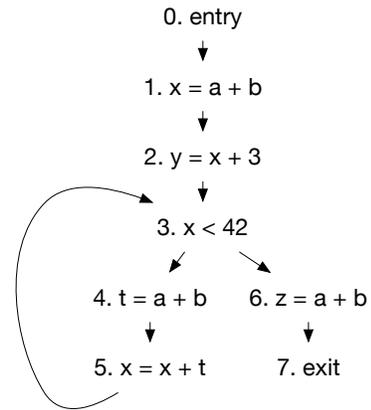
let comp e = comp_expr (mksymtab (vars_of e)) e
```

Question 3. De-Optimization (20 points). Each RubeVM program on the left has had one (and only one) optimization applied to it, as indicated. Write a corresponding program on the right that shows the program *before* the optimization was applied.

Optimized	De-optimized (original) code
<p>Common subexpression elimination</p> <pre>const r0, 42 const r1, 13 add r2, r0, r1 mov r3, r2</pre>	<pre>const r0, 42 const r1, 13 add r2, r0, r1 add r3, r0, r1</pre>
<p>Copy propagation</p> <pre>const r0, 42 add r1, r0, r0 mov r2, r1 add r3, r1, r1 mov r4, r1</pre>	<pre>const r0, 42 add r1, r0, r0 mov r2, r1 add r3, r1, r1 mov r4, r2</pre> <p>(several answers are possible)</p>
<p>Constant Folding</p> <pre>const r0, 42 mk.tab r1 const r2, 84 wr.tab r1, r0, r2</pre>	<pre>const r0, 42 mk.tab r1 add r2, r0, r0 wr.tab r1, r0, r2</pre> <p>(several answers are possible)</p>
<p>Dead code elimination</p> <pre>mk.tab r0 const r1, "a" const r3, 1 wr.tab r0, r1, r3</pre>	<pre>mk.tab r0 const r1, "a" const r2, "b" const r3, 1 wr.tab r0, r1, r3</pre> <p>(several answers are possible)</p>
<p>Loop invariant code motion</p> <pre>const r0, 5 const r1, 10 if_zero r0, 2 sub r0, r0, r1 jmp -3</pre>	<pre>const r0, 5 if_zero r0, 3 const r1, 10 sub r0, r0, r1 jmp -4</pre> <p>(several answers are possible)</p>

Question 4. Data Flow Analysis (30 points).

a. (20 points) In the following table, show each iteration of *available expressions* for the control-flow graph on the right. For each iteration, list the statement taken from the worklist in that step, the value of *out* computed for that statement, and the new worklist at the end of the iteration. You may or may not need all the iterations; you may also add more iterations if needed. Assume $x < 42$ is *not* a possible available expression, and do not add the exit node to the worklist.



Use \emptyset for the set of no expressions, and \top for the set of all expressions. What is \top ?

$\top =$ a+b, x+3, x+t

What are the initial *out*'s for each statement?

Stmt	0	1	2	3	4	5	6
Initial <i>out</i>	\emptyset	\top	\top	\top	\top	\top	\top

Iteration	0	1	2	3	4
Stmt taken from worklist	N/A	1	2	3	4
<i>Out</i> of taken stmt	N/A	a+b	a+b,x+3	a+b,x+3	a+b, x+3
New worklist	1,2,3,4,5,6	2,3,4,5,6	3,4,5,6	4,5,6	5,6

Iteration	5	6	7	8	9
Stmt taken from worklist	5	3	4	5	6
<i>Out</i> of taken stmt	a+b	a+b	a+b	a+b	a+b
New worklist	3, 6	4,6	5,6	6	\emptyset

Iteration	10	11	12	13	14
Stmt taken from worklist					
<i>Out</i> of taken stmt					
New worklist					

b. (10 points) Write the gen and kill sets for the two analyses and statements shown below. Assume the set of expressions is $\{x+y, x+1, a+1, b+1\}$, and the set of variables is $\{a, b, x, y\}$. Write \emptyset for an empty gen or kill set.

stmt	Live vars		Very busy exprs	
	gen	kill	gen	kill
$a := x + y$	x,y	a	$x+y$	$a+1$
$a := b$	b	a	\emptyset	$a+1$
$x := x + 1$	x	\emptyset OR x	$x+1$	$x+y$ OR $x+y, x+1$