Name:

Midterm 2

CMSC 430
Introduction to Compilers
Fall 2016

November 21, 2016

Instructions

This exam contains 7 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		20
3		30
4		20
5		10
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Briefly describe what an *intermediate representation* is.

Answer: An intermediate representation is a program representation that is at a level of abstraction in between the program source code/abstract syntax tree and the compiler output (typically bytecode or machine code).

b. (5 points) List three things that might be stored in an *activation record* or *stack frame.*

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

c. (5 points) Write down the data-flow equations for a *backward may* analysis, i.e., define $\mathsf{in}(s)$ and $\mathsf{out}(s)$ for a statement s in terms of $\mathsf{gen}(s)$ and $\mathsf{kill}(s)$. Write $\mathsf{pred}(s)$ for the predecessor of s and $\mathsf{succ}(s)$ for the successor of s.

$$\mathsf{in}(s) = (\mathsf{out}(s) - \mathsf{kill}(s)) \cup \mathsf{gen}(s)$$

$$\mathsf{out}(s) = \bigcup_{s' \in \mathsf{succ}(s)} \mathsf{in}(s')$$

d. (5 points) Briefly explain what it means for a type system to be *sound*. (Do not simply refer to progress and preservation—define soundness in more direct terms.)

Answer: Soundness means that a well-typed program either evaluates forever ("diverges") or eventually reduces to a value. In other words, no mater how long a well-typed program runs, it will never get *stuck* in the operational semantics sense.

Question 2. Code Generation (20 points). Below is part of the RubeVM instruction set, followed by a type representing expression ASTs for a small language.

```
type reg = [ `Reg of int ]
type value = [ `Int of int | `Str of string |
               `Id of string ]
type id = [ `Id of string ]

type instr =
  | I_const of reg * value (* dst, src *)
  | I_mov of reg * reg (* dst, src *)
  | I_add of reg * reg * reg (* dst, src1, src2 *)
  | I_sub of reg * reg * reg (* dst, src1, src2 *)
  | I_mul of reg * reg * reg (* dst, src1, src2 *)
  | I_div of reg * reg * reg (* dst, src1, src2 *)
  | I_eq of reg * reg * reg (* dst, src1, src2 *)
  | I_lt of reg * reg * reg (* dst, src1, src2 *)
  | I_leq of reg * reg * reg (* dst, src1, src2 *)
  | I_jmp of int (* offset *)
  | I_if_zero of reg * int (* src, offset *)

type expr =
  | EInt of int
  | EAdd of expr * expr
  | EIfPos of expr * expr * expr
```

The EInt and EAdd forms are standard. The form EIfPos(e1, e2, e3) evaluates e1. Then if e1 is positive, it evaluates to e2; otherwise it evaluates to e3. For example, EIfPos(5, 1, 2) evaluates to 1, and EIfPos(0, 1, 2) and EIfPos(−5, 1, 2) both evaluate to 2.

Write a function comp_expr : expr −> reg * (instr list) that returns a list of instructions that, when run on RubeVM, will compute the expression's value and store it in the returned reg. Note you may not evaluate the expression as part of code generation, i.e., you have to write a compiler, not an interpreter. You may write as many helper functions as you need. Your code should be self-contained, e.g., if you need next_reg you need to implement it, though you may use OCaml standard library functions.

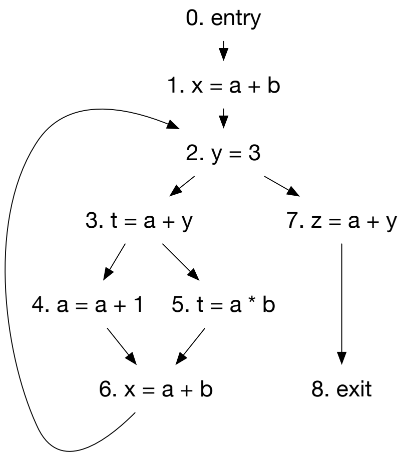
Answer:

```
let next_reg =
  let n = ref 0 in
    fun () -> (let temp = !n in n:=!n+1; `Reg temp)

let rec comp_expr = function
  | EInt n ->
      let r = next_reg () in
        (r, [I_const (`Reg r, `Int n)])
  | EAdd (e1, e2) ->
      let r = next_reg () in
      let (r1, p1) = comp_expr e1 in
      let (r2, p2) = comp_expr e2 in
      (r, p1 @ p2 @ [I_add (r, r1, r2)])
  | EIfPos (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 @
          [I_const (r, `Int 0);
           I_leq (r, r1, r); (* ((r = 1) <=> (r1 <= 0)) and ((r = 0) <=> (r1 > 0)) *)
            I_if_zero (r, (2 + (List.length p3)))] @
          p3 @
          [I_mov (r, r3);
           I_jmp (1 + (List.length p2))] @
          p2 @
          [I_mov (r, r2)]
      )
```

Question 3. Data flow analysis (30 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. Do not add the entry node to the worklist. Note that we branch directly from assignment statements in this CFG to keep the problem shorter.



Use ∅ for the set of no expressions, and ⊤ for the set of all expressions. What is ⊤?

⊤ = $\boxed{\text{a+b, a+y, a+1 (optional), a*b}}$

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

Stmt	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Initial out	∅	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤

Iteration	**0**	**1**	**2**	**3**	**4**
Stmt taken from worklist	N/A	1	2	3	4
out of taken stmt	N/A	a+b	a+b	a+b,a+y	∅
New worklist	1,2,3,4,5,6,7,8	2,3,4,5,6,7,8	3,4,5,6,7,8	4,5,6,7,8	5,6,7,8

Iteration	**5**	**6**	**7**	**8**	**9**
Stmt taken from worklist	5	6	2	7	8
out of taken stmt	a+b,a+y,a*b	a+b	a+b	a+b,a+y	a+b,a,y
New worklist	6,7,8	2,7,8	7,8	8	∅

Iteration	**10**	**11**	**12**	**13**	**14**
Stmt taken from worklist					
out of taken stmt					
New worklist					

Question 4. Optimization (20 points). Fill in the following table so that, from left to right, each row shows a sequence of two optimizations applied to a program written in RubeVM bytecode.

Original code	After copy propagation	After dead code elimination
`const r0, 10`	`const r0, 10`	`const r0, 10`
`const r1, 20`	`const r1, 20`	`const r1, 20`
`add r2, r0, r1`	`add r2, r0, r1`	`add r2, r0, r1`
`mov r3, r2`	`mov r3, r2`	`add r4, r2, r2`
`add r4, r2, r3`	`add r4, r2, r2`	
(only `r4` accessed below here)		

Original code	After constant folding	After common subexpr. elim.
`rd_glob r0, x`	`rd_glob r0, x`	`rd_glob r0, x`
`const r1, 5`	`const r1, 5`	`const r1, 5`
`add r2, r1, r1`	`const r2, 10`	`const r2, 10`
`add r3, r2, r0`	`add r3, r2, r0`	`add r3, r2, r0`
`const r4, 10`	`const r4, 10`	`const r4, 10`
`add r5, r2, r0`	`add r5, r2, r0`	`mov r5, r3`

Original code	After algebraic simplification	After loop inv. code motion
`rd_glob r0, y`	`rd_glob r0, y`	`rd_glob r0, y`
`const r1, 0`	`const r1, 0`	`const r1, 0`
`rd_glob r2, z`	`rd_glob r2, z`	`rd_glob r2, z`
`if_zero r2, 3`	`if_zero r2, 3`	`mov r3, r0`
`add r3, r0, r1`	`mov r3, r0`	`if_zero r2, 2`
`sub r2, r2, r3`	`sub r2, r2, r3`	`sub r2, r2, r3`
`jmp -4`	`jmp -4`	`jmp -3`

Question 5. Type Systems (10 points). Here is the simply typed lambda calculus with integers.

$$e ::= v \mid x \mid e\ e \qquad v ::= n \mid \lambda x\colon t.e \qquad t ::= \mathit{int} \mid t \to t \qquad A ::= \cdot \mid x : t, A$$

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

a. (8 points) Draw a derivation showing that the following term is well-typed in the empty type environment, where we use i instead of *int* to save writing. You need not label the uses of the rules with their names.

$(\lambda f\colon i \to i \to i.f\ 3)\ (\lambda x\colon i.\lambda y\colon i.x)$

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f\colon i \to i \to i \vdash f : i \to i \to i} \qquad \dfrac{}{f\colon i \to i \to i \vdash 3 : i}}{f\colon i \to i \to i \vdash (f\ 3) : i \to i}}{\cdot \vdash (\lambda f\colon i \to i \to i.f\ 3) : (i \to i \to i) \to (i \to i)} \qquad \dfrac{\dfrac{\dfrac{}{y\colon i, x\colon i \vdash x : i}}{x\colon i \vdash \lambda y\colon i.x : i \to i}}{\cdot \vdash (\lambda x\colon i.\lambda y\colon i.x) : i \to i \to i}}{\cdot \vdash (\lambda f\colon i \to i \to i.f\ 3)\ (\lambda x\colon i.\lambda y\colon i.x) : i \to i}$$

b. (2 points) Write down an expression that, according to the type rules above, has type $(i \to i) \to i$.

Answer: There are many possible answers such as $\lambda x\colon i \to i.x\ 3$.