Name:

Directory ID:

University ID:

Midterm 2

CMSC 430
Introduction to Compilers
Fall 2018

Instructions

This exam contains 12 pages, including this one. Make sure you have all the pages. Write your name, directory ID, and university ID number on the top of this page, and write your directory ID at the bottom left of *every* 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		20
4		20
5		20
Total		100

Question 1. Short Answer (20 points).

a. (3 points) Briefly describe the difference between a function call and a system call. Explain whether the two have the same calling convention and why or why not.

b. (3 points) Briefly describe two code generation techniques for compiling C switch statements. Explain how the two techniques compare, i.e., in what situation(s) one technique should be favored over the other.

c. (3 points) Briefly describe symbolic execution and name one application of the technique. What is the fundamental challenge that prevents exhaustive symbolic execution for most real-world programs?

d. (5 points) The following C program contains two functions, `bar()` and `foo()`. `bar()` calls `foo()`. Draw a representation of a 32-bit x86 stack just after the assignment on line 9 is executed. Include as much stack history as you can infer from the code snippet. Clearly indicate the current "top" of the stack (`esp`), the current frame pointer (`ebp`), and which direction the stack grows. You can assume the code is compiled without any optimizations and that `main()` calls `bar()`. If you make any other assumptions, state those as well.

```
1  int bar(int x, int y) {
2      int z;
3      z = foo(x + 10, y - 10);
4      return z;
5  }
6
7  int foo(int a, int b) {
8      int c = 100;
9      int d = a * b + c;
10     // ...
11     return d;
12 }
```

e. (6 points) Translate the following program to 3-address code. Draw the control-flow graph for the resulting program.

```
x = 10
y = 3
z = x + (y * 10) + 100
while ((z > y) && (z > x)) {
    if (a > 10) {
        z = z - 100
    } else {
        z = z - x * 2
    }
}
```

Question 2. Code Generation (20 points). Below (left) is a set of types representing a small machine instruction set, followed by (right) a type representing expression ASTs for a small language.

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

type instr =
  | ILoad of reg * val          (* dst, src *)
  | IStore of id * reg          (* dst, src *)
  | IAdd of reg * reg * reg     (* dst, src1, src2 *)
  | ISub of reg * reg * reg     (* dst, src1, src2 *)
  | IIfZero of reg * int        (* guard, target *)
  | IJmp of int                 (* target *)
  | IMov of reg * reg           (* dst, src *)
```

```
type expr =
  | EInt of int                 (* integers *)
  | EId of string               (* variables *)
  | EAdd of expr * expr         (* addition *)
  | ESub of expr * expr         (* subtraction *)
  | EAssn of string * expr      (* assignment *)
  | ESeq of expr * expr         (* sequences *)
  | EWhile of expr * expr       (* while loops *)
  | EDoWhile of expr * expr     (* do-while loops *)
```

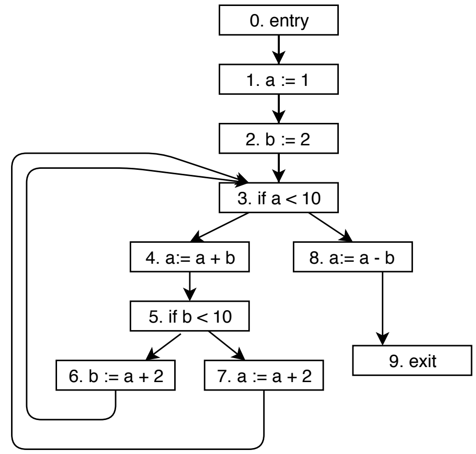
The instruction set has direct support for named locations (i.e., variables), which can be read from or written to via the ILoad and IStore instructions, respectively. ILoad also supports loading a register with a constant integer. IAdd and ISub implement register addition and subtraction, respectively. In both cases, the result is stored in a register. The IJmp (absolute jump) and IIfZero (conditional jump) instructions adjust the PC relative to the current instruction's PC. IMov copies the value stored in one register to another. The machine supports an unlimited number of registers.

The expressions EInt and EId represent constant integers and variables, respectively. EAdd and ESub are the standard binary addition and subtraction expressions. EAssn represents an assignment to a variable. ESeq is the sequence of two expressions. The expression EWhile(e1, e2) executes the body e2 as long as the guard e1 is not zero, and the whole expression evaluates to 0. EDoWhile(e1, e2) executes e1 until e2 becomes non-zero. Note that a do-while 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). The loop itself should evaluate to 0.

Question is on the next page.

Write a function comp_expr : expr → reg ∗ (instr list) that takes a single expression and returns the output register and a list of instructions that compute the expression. You may define as many helper functions as you need, and you may also use OCaml standard library functions.

Question 3. Data Flow (20 points).



In the following table, show each iteration of *reaching definitions* 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.

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

⊤ = ______

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			
out of taken stmt	N/A				
New worklist	1,2,3,4,5,6,7,8				

Iteration	**5**	**6**	**7**	**8**	**9**
Stmt taken from worklist					
out of taken stmt					
New worklist					

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

Question 4. Optimization (20 points). Below are (left) a Simpl function that I wrote to test my project 4 and (right) the RubeVM code that my compiler generated.

```
def foo(x)
  t = mktab();
  i = 0;
  while (i < x) do
    a = x + 10;
    if a < 5 then
      t[i] = a + x
    else
      t[i] = a
    end;
    i = i + 1
  end
end
```

```
foo:
  mov r7, r0
  const r10, mktab
  call r10, 9, 8
  const r11, 0
  lt r12, r11, r7
  if_zero r12, 15
  const r14, 10
  add r13, r7, r14
  const r17, 5
  lt r16, r13, r17
  if_zero r16, 4
  add r18, r13, r7
  wr_tab r9, r11, r18
  mov r15, r18
  jmp 2
  wr_tab r9, r11, r13
  mov r15, r13
  const r20, 1
  add r19, r11, r20
  mov r11, r19
  jmp -17
  const r21, 0
  ret r21
```

In the space below, identify two (2) *local* and two (2) *global* optimizations that could be applied to this code. Clearly identify the type of optimization, the code location, and how the code would change. Hint: you may want to convert the RubeVM code to a more familiar three-address form and draw a control-flow graph first.

a. (5 points) Local optimization 1

b. (5 points) Local optimization 2

c. (5 points) Global optimization 1

d. (5 points) Global optimization 2

Question 5. Type Systems (20 points). Here is the simply typed lambda calculus with integers, floats, and pairs.

$$e ::= v \mid x \mid e\ e \mid (e, e) \qquad v ::= n \mid f \mid \lambda x{:}t.e \qquad t ::= \mathit{int} \mid \mathit{float} \mid t \to t \mid t \times t \qquad A ::= \cdot \mid x : t, A$$

$$\textsc{Int}\ \frac{}{A \vdash n : \mathit{int}} \qquad \textsc{Float}\ \frac{}{A \vdash f : \mathit{float}} \qquad \textsc{Var}\ \frac{}{A \vdash x : A(x)} \qquad \textsc{Pair}\ \frac{A \vdash e_1 : t \qquad A \vdash e_2 : t'}{A \vdash (e_1, e_2) : t \times t'}$$

$$\textsc{Lam}\ \frac{x{:}t, A \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \textsc{App}\ \frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\ e_2 : t'}$$

a. (5 points) Draw a derivation showing that the following term is well-typed in the given type environment, where we use i instead of *int* and f instead of *float* to save space. You need not label the uses of the rules with their names.

$A = +{:}\, i \to i \to i,\ \ \oplus{:}\, f \to f \to f$

$A \vdash ((\lambda x{:}\, i.\lambda y{:}\, f.(+\ x\ 5, \oplus\ y\ 5.0))\ \ 7\ \ 7.0){:}\, i \times f$

b. (5 points) If we make *int* a subtype of *float*, we get the following additional type rules (note that application has been updated):

S-NUM
$$\frac{}{int \leq float}$$

S-PROD
$$\frac{t_1 \leq t_1' \quad t_2 \leq t_2'}{t_1 \times t_2 \leq t_1' \times t_2'}$$

APP
$$\frac{A \vdash e_1 : t_1 \to t_1' \qquad A \vdash e_2 : t_2 \qquad t_2 \leq t_1}{A \vdash e_1\ e_2 : t_1'}$$

S-ARROW
$$\frac{}{t_1 \to t_1' \leq t_2 \to t_2'}$$

i. (2 points) The rule S-ARROW, which extends the subtyping relationship to arrow types (functions), is incomplete. Fill in the rest of the rule.

ii. (3 points) Given your rule, can ($+$: $int \to int \to int$) be used in computations where ($\oplus$: $float \to float \to float$) is expected? Explain why or why not. If not, give an example of a type that can be used where $\oplus$ is expected.

c. (10 points) Consider type inference for the simply-typed lambda calculus, this time without the float, pair, and subtyping extensions.

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

$$
\textsc{Int}\ \frac{}{A \vdash n : \mathit{int}}
\qquad
\textsc{Var}\ \frac{x \in \mathit{dom}(A)}{A \vdash x : A(x)}
\qquad
\textsc{Lam}\ \frac{x{:}\,\alpha, A \vdash e : t' \qquad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t'}
\qquad
\textsc{App}\ \frac{\begin{array}{cc} A \vdash e_1 : t & A \vdash e_2 : t' \\ t = t' \to \beta & \beta \text{ fresh} \end{array}}{A \vdash e_1\ e_2 : \beta}
$$

i. (5 points) Draw a derivation showing type inference applied to the following term. Use subscripts to distinguish new instances of type inference variables. For example, α_1, α_2, β_1, β_2, etc.

$$+ {:}\ \mathit{int} \to \mathit{int} \to \mathit{int} \vdash (\lambda x.+\ \ x\ 5)\ 37$$

ii. (3 points) Draw a union find data structure representing the constraints.

iii. (2 points) Write down a solution to the associated constraints.

This page is intentionally blank for extra work space. If you want the work on this page to count, clearly label which question you are answering and write "see back page" in the answer space for the question.

This page is intentionally blank for extra work space. If you want the work on this page to count, clearly label which question you are answering and write "see back page" in the answer space for the question.