Name:

Midterm 1

CMSC 430
Introduction to Compilers
Spring 2015

Instructions

This exam contains 11 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		30
2		30
3		40
Total		100

Question 1. Short Answer (30 points).

a. (5 points) Briefly explain the role of a *lexer* and the role of a *parser* in a compiler front-end.

Answer: A lexer turns a stream of characters into a stream of tokens, a parser turns a stream of tokens into an abstract syntax tree.

b. (5 points) Draw a line from the compiler concepts on the left to the (single) most fitting theoretical concept on the right that underpins it:

1. lexer
2. parser

1. Turing machines
2. regular expressions
3. lambda calculus
4. control flow graphs
5. discrete-time Markov chains
6. context-free grammars
7. natural semantics

Answer: lexer → regular expressions, parser → context-free grammars.

c. (5 points) Give a concise definition of a *basic block*?

Answer: A basic block is a sequence of 3-address instructions that contains no jumps except possibly the last instruction and has no code that jumps to it except possibly the first instruction.

d. (10 points) Consider the following data type for abstract syntax trees for lambda calculus extended with a form for "let" and "let rec":

```
type expr =
  | Var of string
  | Lam of string * expr
  | App of expr * expr
  (* AST for let x = e1 in e2 *)
  | Let of string * expr * expr
  (* AST for let rec x = e1 in e2 *)
  | LetRec of string * expr * expr
```

Write a function `fvs : expr -> string list` that returns the list of free variables in an expression. Your function should reflect the usual scoping for let and let rec as done in OCaml, for example.

Answer:

```
let rm x xs = List.filter (fun y → x = y) xs
let rec fvs e =
  match e with
  | Var x → [x]
  | Lam (x, e) → rm x (fvs e)
  | App (e0, e1) → (fvs e0) @ (fvs e1)
  | Let (x, e0, e1) → (fvs e0) @ (rm x (fvs e1))
  | LetRec (x, e0, e1) → rm x ((fvs e0) @ (fvs e1))
```

e. (5 points) Give an IMP program that is equivalent to the following program in three-address code:

```
1: z := 10
2: if (z = 0) goto 6
3: b := b * 2
4: z := z + 1
5: goto 2
6: y := 20
```

Answer:

```
z := 10;
while (z != 0) {
  b := b * 2;
  z := z + 1;
}
y := 20;
```

Question 2. Miscellaneous (30 points).

a. (10 points) Translate the following IMP program to three-address code:

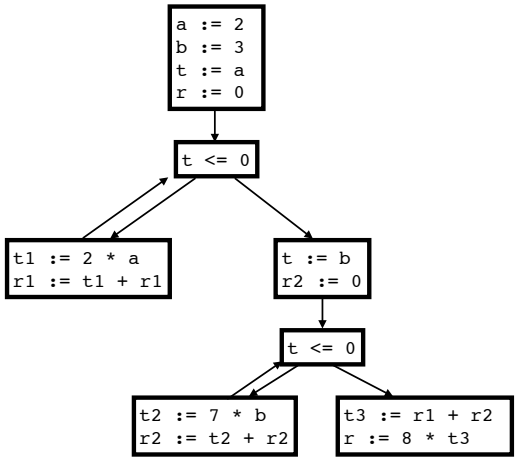
```
a := 2;
b := 3;
t := a;
r1 := 0;
while (t > 0) {
  r1 := (2 * a) + r1;
  t := t - 1;
}
t := b;
r2 := 0;
while (t > 0) {
  r2 := (7 * b) + r2;
  t := t - 1;
}
r := 8 * (r1 + r2);
```

Answer:

```
   a := 2
   b := 3
   t := a
   r1 := 0
1: if (t <= 0) goto 2
   t1 := 2 * a
   r1 := t1 + r1
   goto 1
2: t := b
   r2 := 0
3: if (t <= 0) goto 4
   t2 := 7 * b
   r2 := t2 + r2
   goto 3
4: t3 := r1 + r2
   r := 8 * t3
```

b. (10 points) Draw a control flow graph corresponding to the three-address code you gave in the previous problem.

Answer:



c. (10 points) Here is a grammar. Give a derivation that shows `let x = 5 in let z = 2 in x + z` is in the language:

$$
\begin{array}{rcl}
E & \rightarrow & \texttt{let}X\texttt{=}E\texttt{in}E \\
E & \rightarrow & I \\
E & \rightarrow & X \\
E & \rightarrow & E\texttt{+}E \\
I & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
X & \rightarrow & \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots
\end{array}
$$

Answer:

$$
\begin{array}{rcl}
E & \rightarrow & \texttt{let}\ X\ \texttt{=}\ E\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ E\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ X\ \texttt{=}\ E\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ \texttt{z}\ \texttt{=}\ E\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ \texttt{z}\ \texttt{=}\ \texttt{2}\ \texttt{in}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ \texttt{z}\ \texttt{=}\ \texttt{2}\ \texttt{in}\ E\ \texttt{+}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ \texttt{z}\ \texttt{=}\ \texttt{2}\ \texttt{in}\ \texttt{x}\ \texttt{+}\ E \\
 & \rightarrow & \texttt{let}\ \texttt{x}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \texttt{let}\ \texttt{z}\ \texttt{=}\ \texttt{2}\ \texttt{in}\ \texttt{x}\ \texttt{+}\ \texttt{z}
\end{array}
$$

Question 3. Operational semantics (40 points). Consider lambda calculus extended with integers, updatable references, and sequences. (Here ℓ ("location") is a pointer, and a store S maps locations to values.)

$$\begin{array}{rcl} v & ::= & n \mid \lambda x.e \mid \ell \\ e & ::= & v \mid x \mid e\ e \mid \text{ref } e \mid\ !e \mid\ \ e; e \mid e := e \\ S & ::= & \emptyset \mid S[\ell \mapsto v] \end{array}$$

Here is an operational semantics for this language (it is the same as the practice exam except sequences have been added):

$$\textsc{Beta}\ \frac{}{\langle S, (\lambda x.e_1)v_2\rangle \to \langle S, e_1[v_2/x]\rangle} \qquad \textsc{AppL}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1\ e_2\rangle \to \langle S', e_1'\ e_2\rangle} \qquad \textsc{AppR}\ \frac{\langle S, e_2\rangle \to \langle S', e_2'\rangle}{\langle S, v\ e_2\rangle \to \langle S', v\ e_2'\rangle}$$

$$\textsc{Ref}\ \frac{\ell \notin dom(S) \qquad S' = S[\ell \mapsto v]}{\langle S, \text{ref } v\rangle \to \langle S', \ell\rangle} \qquad \textsc{RefIn}\ \frac{\langle S, e\rangle \to \langle S', e'\rangle}{\langle S, \text{ref } e\rangle \to \langle S', \text{ref } e'\rangle}$$

$$\textsc{Assign}\ \frac{S' = S[\ell \mapsto v]}{\langle S, \ell := v\rangle \to \langle S', v\rangle} \qquad \textsc{AssignL}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1 := v\rangle \to \langle S', e_1' := v\rangle}$$

$$\textsc{AssignR}\ \frac{\langle S, e_2\rangle \to \langle S', e_2'\rangle}{\langle S, e_1 := e_2\rangle \to \langle S', e_1 := e_2'\rangle} \qquad \textsc{Deref}\ \frac{}{\langle S, !\ell\rangle \to \langle S, S(\ell)\rangle} \qquad \textsc{DerefIn}\ \frac{\langle S, e\rangle \to \langle S', e'\rangle}{\langle S, !e\rangle \to \langle S', !e'\rangle}$$

$$\textsc{Seq}\ \frac{}{\langle S, v; e\rangle \to \langle S, e\rangle} \qquad \textsc{SeqIn}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1; e_2\rangle \to \langle S', e_1'; e_2\rangle}$$

a. (10 points) Define a grammar of *contexts*, C, so that the following rule can replace as many of the above rules as possible. List the names of the rules replaced by the following:

$$\frac{\langle S, e\rangle \to \langle S', e'\rangle}{\langle S, C[e]\rangle \to \langle S', C[e']\rangle}\ \textsc{Context}$$

Answer:

$$C ::= \square \mid C\ e \mid v\ C \mid \text{ref } C \mid C := v \mid e := C \mid\ !C \mid C; e$$

Replaces: AppL, AppR, RefIn, AssignL, AssignR, DerefIn, SeqIn.

b. (10 points) The prior semantics reduces assignments right-to-left and reduces applications left-to-right. Modify the semantics either by adding rules to the original definition or modifying your definition of contexts to be ambiguous (that is, allow applications and assignments to reduce left-to-right or right-to-left non-deterministically.)

Answer:

$$C ::= \Box \mid C\ e \mid e\ C \mid \text{ref } C \mid C := e \mid e := C \mid\ !C \mid C; e$$

c. (10 points) Construct an example expression that can produce different results depending on the order of reduction. Give the two different results and describe the circumstances under which each result arises (e.g. "applications are done left-to-right, assignments right-to-left"). You do **not** need to show the steps of reduction.

Answer: $(\lambda x.((x := 1; (\lambda y.y))(x := 2)))(\text{ref } 0)$

If applications are evaluated left-to-right, the result is 2. If applications are evaluated right-to-left, the result is 1.

d. (10 points) Breifly explain the advantages and disadvantages between *call-by-name* and *call-by-value* evaluation strategies.

Answer: Advantages: Call-by-name avoids work if unneeded and always finds a normal form if one exists. Call-by-value never duplicates work by evaluating an argument more than once.

Disadvantages: Call-by-name may duplicate work by evaluating arguments more than once. Call-by-value may do work that is not needed.