Name:

(Practice) Midterm 1

CMSC 430
Introduction to Compilers
Spring 2015

Instructions

This exam contains 9 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		30
3		35
Total		90

Question 1. Short Answer (25 points).

a. (5 points) Briefly explain the difference between a *small step* semantics and *big step* semantics.

> **Answer:** A big step semantics is a relation between programs and values that captures the notion of "evaluation". A small step semantics is relation between programs and programs that captures the notion of "reduces in one step". The latter is useful for reasoning about the steps of computation a program goes through, including programs that error (get "stuck") and run forever.

b. (5 points) What's the difference between an *ahead-of-time* compiler and a *just-in-time* compiler? Explain in at most 1-2 sentences.

> **Answer:** An ahead-of-time compiler translates a program into an equivalent program (in a potentially different language) in a single batch operation. A just-in-time compiler runs the program directly as in an interpreter, but compiles code on the fly as needed and using information gained while running the program.

c. (5 points) In at most a few sentences, explain what it means for a compiler to be *correct*?

Answer:

A compiler from language L_1 to L_2 is correct if it translates a program P written in L_1 to a program Q in L_2 such that interpreting Q produces an answer equivalent to interpreting P.

d. (5 points) Why do compilers sometimes need to introduce *temporary variables* when translating to an intermediate representation?

Answer: If the IR is three-address code, then temporary variables are needed to hold intermediate computation results when executing complex expressions.

e. (5 points) Give three-address code equivalent to the following Imp program:

```
n := 2;
m := 64;
r := 1;

while (m > 0) {
  s := r
  t := 0
  while (s > 0) {
    t := t + n;
    s := s − 1;
  }
  r := t;
  m := m − 1;
}
```

Answer:

```
 1: n := 2
 2: m := 64
 3: r := 1
 4: if m <= 0 goto 14
 5: s := r
 6: t := 0
 7: if s <= 0 goto 11
 8: t := t + n
 9: s := s − 1
10; goto 7
11: r := t
12: m := m − 1
13: goto 4
14: skip
```

Question 2. Miscellaneous (40 points).

a. (10 points)

Below is a small step operation semantics for boolean expressions that "short-circuits" evaluation—it does not reduce sub-expression that are unneeded:

$$\begin{array}{rcl} bv & ::= & \mathit{true} \mid \mathit{false} \\ b & ::= & bv \mid b \wedge b \mid b \vee b \mid \neg b \end{array}$$

$$\frac{}{\mathit{true} \vee b \rightarrow \mathit{true}}\ \textsc{OrTrue} \qquad \frac{}{\mathit{false} \vee b \rightarrow b}\ \textsc{OrFalse} \qquad \frac{}{\mathit{true} \wedge b \rightarrow b}\ \textsc{AndTrue}$$

$$\frac{}{\mathit{false} \wedge b \rightarrow \mathit{false}}\ \textsc{AndFalse} \qquad \frac{b_1 \rightarrow b_1'}{b_1 \wedge b_2 \rightarrow b_1' \wedge b_2}\ \textsc{AndStep} \qquad \frac{b_1 \rightarrow b_1'}{b_1 \vee b_2 \rightarrow b_1' \vee b_2}\ \textsc{OrStep}$$

$$\frac{}{\neg\mathit{true} \rightarrow \mathit{false}}\ \textsc{NegTrue} \qquad \frac{}{\neg\mathit{false} \rightarrow \mathit{true}}\ \textsc{NegFalse} \qquad \frac{b \rightarrow b'}{\neg b \rightarrow \neg b'}\ \textsc{NegStep}$$

Define a grammar of *contexts*, C, such that the following rule can replace some of the above rules. List the names of rules it replaces.

$$\frac{b \rightarrow b'}{C[b] \rightarrow C[b']}\ \textsc{Context}$$

Answer:

$$C ::= [\,] \mid C \wedge b \mid C \vee b \mid \neg C$$

b. (10 points) Here is a grammar, give a deriviation showing that *abbcd* is in the language A:

$$\begin{array}{rcl} A & \rightarrow & aBd \\ B & \rightarrow & bB \\ B & \rightarrow & \varepsilon \\ B & \rightarrow & c \end{array}$$

Answer:

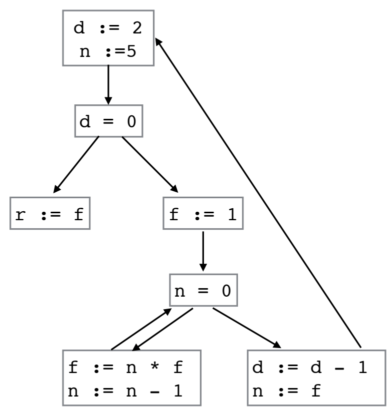
$$A \rightarrow aBd \rightarrow abBd \rightarrow abbBd \rightarrow abbcd$$

c. (10 points) Here is a program in IR for computing the double factorial of 5 (5!!):

```
 0: d := 2
 1: n := 5
 2: if (d=0) goto 11
 3: f := 1
 4: if (n=0) goto 8
 5: f := n * f
 6: n := n - 1
 7: goto 4
 8: d := d - 1
 9: n := f
10: goto 2
11: r := f
```

Draw the control flow graph for this program.

Answer:



Question 3. Operational semantics (35 points). Consider lambda calculus extended with integers and updatable references. (Here ℓ ("location") is a pointer, and a store S maps locations to values.)

$$\begin{array}{rcl} v & ::= & n \mid \lambda x.e \mid \ell \\ e & ::= & v \mid x \mid e\ e \mid \text{ref } e \mid !e \mid e_1 := e_2 \\ S & ::= & \emptyset \mid S[\ell \mapsto v] \end{array}$$

Here is most of an operational semantics for this language:

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

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

$$\textsc{Assign}\ \frac{S' = S[\ell \mapsto v]}{\langle S, \ell := v\rangle \to \langle S', v\rangle} \qquad \textsc{AssignL}\ \frac{\langle S, e_1\rangle \to \langle S', e_1'\rangle}{\langle S, e_1 := v\rangle \to \langle S', e_1' := v\rangle} \qquad \textsc{AssignR}\ \frac{\langle S, e_2\rangle \to \langle S', e_2'\rangle}{\langle S, e_1 := e_2\rangle \to \langle S', e_1 := e_2'\rangle}$$

a. (5 points) In this semantics, is the left-hand side of an assignment evaluated first; is the right-hand side evaluated first; or is the choice non-deterministic? Explain your answer briefly.

> **Answer:** The right-hand side is evaluated first, as can be seen in Rule AssignL, which requires the right hand side be fully evaluated.

b. (10 points) Let $\emptyset$ be the empty store. Show a derivation that in this operational semantics, the reduction at the bottom holds. (You can draw your derivation up, above the reduction.)

> **Answer:**

$$\cfrac{\cfrac{\langle \emptyset, \text{ref } 42\rangle \to \langle [\ell \mapsto 42], \ell\rangle}{\langle \emptyset, (\lambda x.x)\ (\text{ref } 42)\rangle \to \langle [\ell \mapsto 42], (\lambda x.x)\ \ell\rangle}}{\langle \emptyset, ((\lambda x.x)\ (\text{ref } 42))\ (\lambda z.z)\rangle \to \langle [\ell \mapsto 42], ((\lambda x.x)\ \ell)\ (\lambda z.z)\rangle}$$

$$\langle \emptyset, ((\lambda x.x)\ (\text{ref } 42))\ (\lambda z.z)\rangle \qquad \to \qquad \langle [\ell \mapsto 42], ((\lambda x.x)\ \ell)\ (\lambda z.z)\rangle$$

c. (10 points) Evaluate the following configuration until no more reductions are possible. For this part, just show the reduction steps and not the derivations of the steps. Use ℓ_1, ℓ_2, ℓ_3, etc if you need more locations. (Note this example will use a pointer to a pointer, which is allowed.)

$$\langle \emptyset, ((\lambda x.x)(\text{ref } 42)) := ((\lambda y.\lambda z.y)\ (\text{ref } 43)\ 44)\rangle \qquad \rightarrow$$

Answer:

$$\begin{aligned}
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := ((\lambda y.\lambda z.y)\ \ell_1\ 44)\rangle &\qquad \rightarrow \\
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := ((\lambda z.\ell_1)\ 44)\rangle &\qquad \rightarrow \\
\langle [\ell_1 \mapsto 43], ((\lambda x.x)(\text{ref } 42)) := \ell_1\rangle &\qquad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto 42], ((\lambda x.x)\ell_2) := \ell_1\rangle &\qquad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto 42], \ell_2 := \ell_1\rangle &\qquad \rightarrow \\
\langle [\ell_1 \mapsto 43, \ell_2 \mapsto \ell_1], \ell_1\rangle &
\end{aligned}$$

d. (10 points) Write the missing operational semantics rule(s) for dereference (written $!e$).

Answer:

$$\textsc{Deref}\ \frac{}{\langle S, \text{ref } \ell\rangle \rightarrow \langle S, S(\ell)\rangle} \qquad \textsc{DerefIn}\ \frac{\langle S, e\rangle \rightarrow \langle S', e'\rangle}{\langle S, !e\rangle \rightarrow \langle S', !e'\rangle}$$