

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.

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.

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

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

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;
}
```

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:

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

$$\begin{array}{ccc} \frac{}{true \vee b \rightarrow true} \text{ORTRUE} & \frac{}{false \vee b \rightarrow b} \text{ORFALSE} & \frac{}{true \wedge b \rightarrow b} \text{ANDTRUE} \\ \frac{}{false \wedge b \rightarrow false} \text{ANDFALSE} & \frac{b_1 \rightarrow b'_1}{b_1 \wedge b_2 \rightarrow b'_1 \wedge b_2} \text{ANDSTEP} & \frac{b_1 \rightarrow b'_1}{b_1 \vee b_2 \rightarrow b'_1 \vee b_2} \text{ORSTEP} \\ \frac{}{\neg true \rightarrow false} \text{NEGTRUE} & \frac{}{\neg false \rightarrow true} \text{NEGFALSE} & \frac{b \rightarrow b'}{\neg b \rightarrow \neg b'} \text{NEGSTEP} \end{array}$$

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.

$$\frac{b \rightarrow b'}{C[b] \rightarrow C[b']} \text{CONTEXT}$$

b. (10 points) Here is a grammar, give a derivation showing that $abcd$ is in the language A :

$$\begin{aligned} S &\rightarrow aBd \\ B &\rightarrow bB \\ B &\rightarrow \varepsilon \\ B &\rightarrow c \end{aligned}$$

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.

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.)

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

Here is most of an operational semantics for this language:

$$\begin{array}{c} \text{BETA} \\ \hline \langle S, (\lambda x.e_1)v_2 \rangle \rightarrow \langle S, e_1[x \mapsto v_2] \rangle \end{array} \quad \begin{array}{c} \text{APPL} \\ \hline \frac{\langle S, e_1 \rangle \rightarrow \langle S', e'_1 \rangle}{\langle S, e_1 e_2 \rangle \rightarrow \langle S', e'_1 e_2 \rangle} \end{array} \quad \begin{array}{c} \text{APPR} \\ \hline \frac{\langle S, e_2 \rangle \rightarrow \langle S', e'_2 \rangle}{\langle S, v e_2 \rangle \rightarrow \langle S', v e'_2 \rangle} \end{array}$$

$$\begin{array}{c} \text{REF} \\ \hline \frac{\ell \notin \text{dom}(S) \quad S' = S[\ell \mapsto v]}{\langle S, \text{ref } v \rangle \rightarrow \langle S', \ell \rangle} \end{array} \quad \begin{array}{c} \text{REFIN} \\ \hline \frac{\langle S, e \rangle \rightarrow \langle S', e' \rangle}{\langle S, \text{ref } e \rangle \rightarrow \langle S', \text{ref } e' \rangle} \end{array}$$

$$\begin{array}{c} \text{ASSIGN} \\ \hline \frac{S' = S[\ell \mapsto v]}{\langle S, \ell := v \rangle \rightarrow \langle S', v \rangle} \end{array} \quad \begin{array}{c} \text{ASSIGNL} \\ \hline \frac{\langle S, e_1 \rangle \rightarrow \langle S', e'_1 \rangle}{\langle S, e_1 := v \rangle \rightarrow \langle S', e'_1 := v \rangle} \end{array} \quad \begin{array}{c} \text{ASSIGNR} \\ \hline \frac{\langle S, e_2 \rangle \rightarrow \langle S', e'_2 \rangle}{\langle S, e_1 := e_2 \rangle \rightarrow \langle S', e_1 := e'_2 \rangle} \end{array}$$

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.

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.)

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

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.)

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

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