Midterm II
CS164, Spring 2004

April 13, 2004

- Please read all instructions (including these) carefully.
- **Write your name, login, SID, and circle the section time.**
- There are 8 pages in this exam and 4 questions, each with multiple parts. Some questions span multiple pages. All questions have some easy parts and some hard parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two pages of handwritten notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We might deduct points if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

LOGIN: ____________________

NAME: ____________________

SID: ____________________

Circle the time of your section: Tue 3:00 Tue 4:00 Wed 10:00 Wed 11:00 Wed 1:00 Wed 2:00

Problem	Max points	Points
1	15	
2	40	
3	20	
4	25	
TOTAL	100	

1 Local Optimizations (15 points)

(a) Consider the following code:

```
x := 2
y := x * 2
z := x + y
w := x * 2
z := z + w
```

Here are three optimized versions of this code. The modified portion of the code is shown with a box around it.

i. x := 2
y := x * 2
z := x + y
w := $\boxed{\texttt{y}}$
z := z + w

ii. x := 2
y := $\boxed{\texttt{2}}$ * 2
z := $\boxed{\texttt{2}}$ + y
w := $\boxed{\texttt{2}}$ * 2
z := z + w

iii. x := 2
y := $\boxed{\texttt{x + x}}$
z := x + y
w := $\boxed{\texttt{x + x}}$
z := z + w

In each version of the code, we have applied one of the optimization techniques discussed in lecture. Write to the right of each block of code what optimization was used.

(b) Consider the following code:

```
a := y
x := a + b
y := x
z := a * y
y := y + x
w := y + x
```

Apply copy propagation to this code. Write the resulting code to the right of the original code.

(c) i. The following block of code makes use of five variables: a, b, c, d, and e. However, we have erased many of these variable references from the original program. In the right-hand column, we provide the results of liveness analysis (i.e., the variables that are live at each program point). Please fill in each blank with a **variable** so that the program is consistent with the results of liveness analysis.

Please note that there are **no dead instructions** in this program. You will need this information to fill in some of the blanks correctly!

```
Code                         Live Variables

                             {a, b, c}
□    :=   □ + a
                             {a, c, d}
□    :=   c + □
                             {c, d}
□    :=   □ + d
                             {e}
□    :=   □ * □
                             {e}
□    :=   42
                             {a, e}
c    :=   □ * e
                             {c, e}
print     □
                             {c}
print     □
                             {}
```

2 Global Analysis (40 points)

The goal of this exercise is to design a global analysis for computing, for each program point and for each variable x, a set of variables that have the same value as x at the given program point. We use the names $E_{in}(x, s)$ and $E_{out}(x, s)$ for the sets of variables discovered to be equal to x before and after instruction s. We assume that these sets **include** x itself.

(a) Explain how can you optimize the instruction $x := y$ if you know E_{in} for this instruction.

(b) Consider an instruction s with two predecessors p_1 and p_2. Write the formula for $E_{in}(x, s)$.

Consider the instruction s_1 of the form $x := y$.

(c) Assuming that $E_{in}(y, s_1) = \{y, z, w\}$, what are $E_{out}(x, s_1)$ and $E_{out}(y, s_1)$?

(d) Assuming that $E_{in}(v, s_1) = \{v, z, x\}$, what is $E_{out}(v, s_1)$ (v and z are some variables other than x or y)?

(e) Write the value $E_{out}(x, s_1)$ as a function of E_{in}.

(f) Write the value of $E_{out}(v, s_1)$ (v is a variable other than x or y) as a function of E_{in}.

(g) What are the values of E that play the same role as # and $*$ from the constant propagation analysis from the lecture?

(h) Is the computation of E a forward or backward analysis?

(i) How many different values can $E_{out}(x, s)$ take, assuming that the program has T variables?

(j) Explain why this global analysis will terminate. What is the maximum number of steps that this analysis will take, for a program with N instructions and T temporary variables? We consider one step to be the update of one of the E_{out} values.

(k) Is there a code fragment with a program point where x and y have *different* values on some execution, yet this analysis discovers that they are equal? Explain your answer, and show a code example if there is one.

(l) Is there a code fragment with a program point where x and y always have the *same* value, yet this analysis does not discover that they are equal? Explain your answer, and show a code example if there is one.

3 Typing and Operational Semantics (20 points)

Cool has a `let` construct with the following typing and operational semantics:

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \\ O[T_0/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \texttt{let}\ x : T_0 \leftarrow e_1\ \texttt{in}\ e_2 : T_2} \qquad \frac{\begin{array}{l} so, S, E \vdash e_1 : v_1, S_1 \\ l_1 = newloc(S_1) \\ S_2 = S_1[v_1/l_1] \\ so, S_2, E[l_1/x] \vdash e_2 : v_2, S_3 \end{array}}{so, S, E \vdash \texttt{let}\ x \leftarrow e_1\ \texttt{in}\ e_2 : v_2, S_3}$$

Note that the static type of the initializer (T_1) can be a subtype of the declared variable type (T_0). Now we introduce a `let_dynamic` construct that is more flexible: the initializer's dvnamic type is checked at *run-time* to be conforming to the declared type. If the value of the initializer is `void` or if the conformance check fails then a run-time error occurs (similar to when a dispatch on `void` is attempted). For example, the following two Cool expressions are both well-typed and run without an error assuming that B is a subtype of A:

```
let_dynamic x : A <- (let y : B <- new B in y) in ...
let_dynamic x : B <- (let y : A <- new B in y) in ...
```

(a) Consider the expression `let_dynamic` $x : T_0 \leftarrow e_1$ `in` e_2. Assuming that this construct evaluates without error, write down the relationship that must exist between the *dynamic type* of the result of `e1` (call that D_1) and T_0.

(b) Using as a reference the typing rule for `let`, fill in the missing parts of the typing rule for `let_dynamic`:

$$\frac{\qquad\qquad\qquad\qquad\qquad\qquad}{O, M, C \vdash \texttt{let\_dynamic}\ x : T_0 \leftarrow e_1\ \texttt{in}\ e_2 :}$$

(c) And now fill in the missing parts of the operational semantic rule for `let_dynamic`. The rule must express the necessary run-time checks but need not consider what happens if the check fails.

$$\frac{}{so, S, E \vdash \texttt{let\_dynamic}\ x : T_0 \leftarrow e_1\ \texttt{in}\ e_2 :}$$

(d) Consider again the expression `let_dynamic` $x : T_0 \leftarrow e_1$ `in` e_2. What relationship must exist between the *static type* of the initializer expression (call that T_1) and T_0 such that there is any chance that the `let_dynamic` executes without a run-time error?

(e) Consider T_1 defined as in the previous point. Is there a static condition between T_0 and T_1 that determines in advance whether `let_dynamic` will execute without a run-time error? Write the condition and argue why it has the required effect, or argue that there is no such condition.

4 Short Answers (25 points)

(a) Give an example of an error in a Cool expression that can be detected during semantic analysis but cannot be detected during parsing.

(b) What does "flow sensitive property" mean?

(c) Is liveness analysis a forward or backward analysis?

(d) Is it possible to design a program analysis for Cool that terminates and identifies precisely those attributes of type `Int` that have only positive values throughout the execution of a program? Justify your answer.

(e) Why is the following claim false: "Copy propagation is useless because the resulting code is just as large and performs just as many operations as the original code"?

(f) Consider the basic block containing the instructions "x := a + b" and "y := a + b" separated by some other instructions. Assuming that this block is not in single assignment form, what restrictions must be placed on the code separating the two instructions to allow common subexpression elimination on the second instruction?

(g) We discussed in lecture cases when a control-flow graph cannot be written in single assignment form. Show such an example.