Midterm II — CS 4610, Spring 2014

- **Write your name and UVa ID on the exam.** Pledge the exam before turning it in.
- There are 15 pages in this exam (including this one) and 9 questions, each with multiple sub-questions.
- You have up to 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two page-sides of 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.
  - *Good Writing Example:* Python and Ruby have implemented some Smalltalk-inspired ideas with a more C-like syntax.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit subquestion blank or write **"no answer"** for a sub-question (e.g., 1a or 3b) you will receive **one-third of the points** for that sub-question (rounded down) since you did not waste our time. If you randomly guess and throw words at us, we will be significantly less sanguine.

UVa ID: KEY

NAME (print): KEY

UVa ID: (yes, again!) KEY

Problem	Max	Your Points
1 — Type Checking and Dispatch	15	
2 — Adding New Expressions	20	
3 — Optimization	15	
4 — Exceptions	10	
5 — Automatic Memory Management	10	
6 — Debugging, Profiling, Native	10	
7 — Code Generation	10	
8 — Linking	5	
9 — Game Theory	5	
Extra Credit	0	
TOTAL	100	

Honor Pledge:

How do you think you did? ______________________________

1 Type Checking and Dispatch (15 points)

Consider the following *incorrect* typing judgment for the double-`SELF_TYPE` case of dynamic dispatch:

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 = \texttt{SELF_TYPE}_C \\ M(C, f) = (T'_1, \ldots, T'_n, \texttt{SELF_TYPE}) \\ \forall\ 1 \leq i \leq n.\ T'_i \leq T_i \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_0} \text{ wrong}$$

In addition, consider this Cool code:

```
class Animal {
  population : Int ;
  addPopulation( x : Int ) : SELF_TYPE { { population <- population + x ; self ; } } ;
  react( z : Object) : SELF_TYPE { ... case z of ... self } ;
};
class Unicorn inherits Animal {
  magic : Int ;
  moreMagic() : SELF_TYPE { ... self } ;

  selfTest() : Object {
    (* your code will go here *)
  }
};
```

(a) *[7 pts]* The modified typing rule wrong is *too strict*: it rejects good programs that are accepted by the normal typing rules. Give Cool code for the body of `selfTest` above that is accepted by the normal typing rules but is rejected by rule wrong.

The rule should have $T_i \leq T'_i$. Instead, argument subtyping is checked the wrong way. Note that you must use "self.meth()" where "meth" returns SELF_TYPE for this rule to be applicable.

self.react(new Int)

(b) *[8 pts]* The modified typing rule wrong is also *unsound*: it allows programs that will lead to run-time errors. Give Cool code for the body of `selfTest` above that is accepted by rule wrong but that is rejected by the normal typing rules.

self.addPopulation(new Object)

2 Adding New Expressions (20 points)

We would like to add a `with` expression to Cool. Inspired by languages like Pascal, the `with` expression allows us to access the fields (attributes) of one object from within another object. Consider this example:

```
class Counter {
  x : Int ;
  getX () : Int { x } ;
} ;
class Main {
  main() : Object { {
    let c : Counter <- new Counter in
    out_int( c.getX() ) ;                          -- output: 0
    let delta : Int <- 2 in
    with x from c do                               -- x lives in c
      x <- x + delta                               -- x is in scope here
    htiw ;
    out_int( c.getX() ) ;                          -- output: 2
  } } ;
} ;
```

The expected output is 0 followed by 2. Informally, `with` a `from` e_0 `do` e_1 `htiw` checks that e_0 has a field named a and then evaluates e_1 with that a added to the local environment, returning the result of that evaluation. It is a type error if the static type of a is `SELF_TYPE`. There is some conceptual overlap between `let` and `with`, although `with` does not allocate new space.

Recall that for every class C the object environment O_C gives the types of all fields (attributes) of C (including any inherited attributes). In the example above, $O_{\texttt{Counter}}(\texttt{x}) = \texttt{Int}$.

(a) *[7 pts]* Complete the typing rule for `with`. (Be careful about `SELF_TYPE`.)

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ T = \text{if } T_0 = \texttt{SELF_TYPE} \text{ then } C \text{ else } T_0 \\ O_T(a) = T_a \\ T_a \neq \texttt{SELF_TYPE} \\ O' = O[a/T_a] \\ O', M, C \vdash e_1 : T_1 \end{array}}{O, M, C \vdash \texttt{with } a \texttt{ from } e_0 \texttt{ do } e_1 \texttt{ htiw } : T_1} \text{ with} - \text{typecheck}$$

(b) *[7 pts]* Complete the operational semantics rule for `with`.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_0 : v_0, S_2 \\ v_0 = X(a_1 = l_1, \ldots, a = l, \ldots, a_n = l_n) \\ E' = E[a/l] \\ so, S_2, E' \vdash e_1 : v_1, S_3 \end{array}}{so, S_1, E \vdash \texttt{with}\ a\ \texttt{from}\ e_0\ \texttt{do}\ e_1\ \texttt{htiw}\ : v_1, S_3} \text{with} - \text{opsem}$$

Now we would like to add a `foreach` iterator expression to Cool. Cool does not have lists, so programmers must write the iteratees directly. Informally, `foreach` x `in` $e_1, \ldots, e_n$ `do` e_{body} `hcaerof` evaluates e_{body} serially n times in a row. For example, `foreach x in 1, 2, 7-4 do out_int(x) hcaerof` outputs `123`. Just before the ith iteration starts, the expression e_i is evaluated and the result is bound to the variable x for use in e_{body} (as in `let`). If $n = 0$, a `foreach` expression returns `void`.

$$\frac{}{so, S_1, E \vdash \texttt{foreach}\ x\ \texttt{in}\quad \texttt{do}\ e_{body}\ \texttt{hcaerof}\ : void, S_1} \text{foreach} - \text{none}$$

(c) *[6 pts]* Complete the remaining operational semantics rule for `foreach`. (You may not use `let` or `;` as a hypothesis in your answer.)

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ l = newloc(S_2) \\ E' = E[\texttt{x}/l] \\ S_3 = S_2[l/v_1] \\ so, S_3, E' \vdash e_{body} : v_{body}, S_4 \\ so, S_4, E \vdash \texttt{foreach}\ x\ \texttt{in}\ e_2, \ldots, e_n\ \texttt{do}\ e_{body}\ \texttt{hcaerof}\ : v_{final}, S_5 \end{array}}{so, S_1, E \vdash \texttt{foreach}\ x\ \texttt{in}\ e_1, \ldots, e_n\ \texttt{do}\ e_{body}\ \texttt{hcaerof}\ : void, S_5} \text{foreach} - \text{some}$$

3 Optimization (15 points)

(a) *[9 pts]* The following block of code makes use of five variables: `a`, `b`, `c`, `d`, and `e`. However, we have erased many of the 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 single **variable** so that the program is consistent with the results of liveness analysis.

Please note that there are **no dead instructions** in this program. (This means only that `X` is always live right after each assignment to `X`; it doesn't mean that you couldn't personally think of some optimizations to apply here.) You will need this information to fill in some of the blanks correctly!

Code	**Live Variables**
	`{a, b, c}`
`[b] := [a] - b`	
	`{b, c}`
`[e] := b + [c]`	
	`{b, e}`
`[d] := [e] + 1`	
	`{b, d}`
`b := [b]`	
	`{b, d}`
`[c] := 123`	
	`{b, c, d}`
`[a] := b + [c]`	
	`{a, d}`
`print [d]`	
	`{a}`
`print [a]`	
	`{}`

(b) *[6 pts]* Draw a control-flow graph for the following code. Each node in your control-flow graph should be a *basic block*. Do not worry about static single assignment form. Every statement in the code should appear somewhere in your control-flow graph.

```
          START
          a <- 11
          if (b < 22) then { goto albus }
          c <- 33
          d <- 44
severus:  e <- 55
          if (f > 66) then {
                g <- 77
                goto severus
          } else {
                h <- 88
          }
albus:    i <- 99
          j <- 0
          END
```

```
        BEGIN
        a <- 11
   /--- if b < 22 then
   |       |
   |       v
   |    c <- 33
   |    d <- 44
   |       |
   |       |  /--------------------\
   |       |  |                    |
   |       v  v                    |
   |    e <- 55                    |
   |    if f > 66 then  ------>  g <- 77
   |       |
   |       v
   |    h <- 88
   |       |
   |       v
   \--> i <- 99
        j <- 0
        END
```

4 Exceptions (10 points)

(a) *[2 pts]* Give one example of an exception that is the clear result of a mistake in a program. Then give one example of an exception that might be raised even in a perfect program.

Mistake: array out of bound, null pointer dereference, etc.

Environmental: disk full, network failure, etc.

(b) *[4 pts]* Explain where and why the least-upper-bound operator $\sqcup$ is used in our formal treatment of language-level exception handling.

Type checking is static and conservative. Since we cannot be certain at compile-time whether or not exceptions will be thrown at run-time, "try A catch B" could either return "A" or "B". So we use the lub, $\sqcup$, just as we do for if-then-else, to conservatively approximate the best type that describes both "A" and "B".

(c) *[4 pts]* Consider the following *incorrect* incomplete typing rule for `try-finally`.

$$\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2 \quad T_1 \leq T_2}{O, M, C \vdash \texttt{try } e_1 \texttt{ finally } e_2 : T_2} \text{ wrong}$$

Give a Cool *expression* that does not typecheck using this rule but would work correctly at run-time.

The problem with the typing rule is that it requires $T_1 \leq T_2$ (for no reason). So any counter-example with $T_1 > T_2$ works:

```
try new Object finally new Int
```

This works fine at run-time (and should have static type Int), but the wrong rule above fails to accept it.

5 Automatic Memory Management (10 points)

(a) *[2 pts]* What can happen to a program that mistakenly frees memory too early and then later tries to access it?

"Anything." The behavior of the program of the program will be undefined. If you're lucky, the program will immediately crash. If you're unlucky, you will experience silent data corruption that does not show up until later, complicating debugging. If you're very unlucky, more exotic symptoms of the same basic problem are entirely possible.

(b) *[3 pts]* Suppose that you have already decided to use garbage collection. Why is *Stop and Copy* a poor choice for C and C++ programs? Why does *Mark and Sweep* work better?

Stop and Copy is a poor choice for C and C++ programs because it requires moving all pointers. In C/C++ one cannot statically identify which values are pointers (and thus should be changed) and which are not. If the GC tries to move a value that is not a pointer it will change the semantics of the program.

Mark and Sweep works better because one can conservatively identify which values are pointers, but the only penalty for being wrong is failing to collect some garbage.

(c) *[3 pts]* Consider the following program:

```
while not_done() {
  ptr = malloc(100 * MEGABYTE);
  do_work(ptr);
  /* done with ptr */
}
```

You are running this program with 4 gigabytes of physical memory and want to use automatic memory management. Would you choose *Mark and Sweep* or *Stop and Copy*? Why?

Mark and Sweep takes time proportional to all of physical memory (live or dead, reachable or unreachable). Stop and Copy cuts available memory in half, but takes time proportional only to reachable (live) memory. Both garbage collectors are invoked when the system runs out of memory.

Suppose we execute the loop 80 times (the number doesn't matter, "80" is just to have a concrete example).

With Mark and Sweep, after every 40 iterations we will exhaust memory (40 * 100 MB = 4 GB) and invoke the GC. So that will happen twice (80/40 = 2). Each invocation of the M&S GC takes time proportional to all of memory. So we take time proportional to 2*4 GB = 8 GB.

With Stop and Copy, after every 20 iterations we will exhaust memory (20 * 100 MB = 2 GB, and Stop and Copy cuts available memory in half) and invoke the GC. So that will happen four times (80/20 = 4). Each invocation of the S&C GC takes time proportional to used memory, which in this example is at most 100 MB (becaus we're done with each pointer after using it once, as per the pseudocode above). So we take time proportional to 4 * 100 MB = 400 MB = 0.4 GB.

So Stop and Copy is 20 times faster than Mark and Sweep in this example (even though it cuts available memory in half and thus collects more frequently)!

(d) *[2 pts]* Name two specific disadvantages of *Reference Counting*. (Be specific. Just saying that it adds some overhead compared to manual management, for example, is not adequate.)

Con: Reference counting will not detect unused cyclic data structures and will thus fail to reclaim memory used for them.

Con: Reference counting requires a non-trivial space overhead. each object must have an associated counter which must typically be at least one machine word in size.

Con: Reference counting can introduce a non-trivial time delay at assignment statements. For example, in "linked_list := NULL", the reference counter may have to traverse the entire linked list, recursively freeing each cell.

Not needed, but just for reference:

Pro: Reference counting does not require moving objects or pointers in memory.

Pro: Reference counting typically does not introduce large "stop the world" pauses, and is thus often favored for real-time programs.

6 Debugging, Profiling, Native (10 points)

(a) *[2 pts]* Name one advantage of *sampling*-based profiling. Then name one disadvantage.

Pro: Sampling is very easy to implement: arrange for the OS to deliver a periodic timer signal and append the current value of the PC to a buffer.

Pro: Sampling introduces very little overhead, and is thus less likely to disturb the behavior of the system being profiled.

Con: Sampling may miss periodic behavior. For example, if you sample every k seconds, you will miss an event that occurs at time $0.5 + nk$ for all n.

Con: Sampling typically only reports the value of the program counter; it does not provide rich information such as the number of objects allocated or the number of times a variable was accessed. To put it another way, sampling indicates where your program spent time, not how often your program performed various actions of interest.

(b) *[2 pts]* When during debugging is operating system intervention required? What bad things would happen if the operating system were not required?

Operating system intervention is required to "attach" the debugger process to the buggy program process (so that the debugger process can control what happens when the subject receives signals or encounters breakpoints). The operating system must check that you have permission to debug the process (typically you must be the owner/creator of that process).

If no such check were performed, you could attach a debugger to any process — such as the SSH daemon or another user's private shell — and then inspect the values of variables and otherwise control its execution. Without this check, any non-privileged user could immediately become root.

(c) *[2 pts]* Name a code optimization and explain why it would complicate debugging.

Dead code elimination complicates "single stepping" because the user's perception of what constitutes a single step will not match the generated assembly code (e.g., what does it mean to single step over code that was eliminated?).

Copy propagation (if you like, coupled with dead code elimination) complicates "variable inspection" because the value of a variable may not actually be available, even if it appears to be in scope in the source code.

Register allocation complicates "variable inspection" because two local variables that do not interfere may be assigned to the same register and thus you may not be able to inspect both at once.

Function inlining complicates "put a breakpoint at the start of this function" because the program may execute the inlined body of the function without actually calling the official function body code.

At a high level, any optimization that reorders code, reduces code, or replaces code with something semantically equivalent complicates debugging because the user may want to step through or inspect "what is shown in the source code".

(d) *[4 pts]* List one advantage of a native code interface. Then list three potential disadvantages or bugs associated with native code interfaces.

Pro: A native code interface allows computationally intensive kernels to be executed with high performance even in a slower "safe" or "scripting" language.

Pro: A native code interface allows a language to take advantage of the wealth of existing third-party and open source libraries (e.g., for graphics, compression, mathematics, networking, etc.).

Pro: A native code interface allows program components to be written in multiple languages while sharing data and code in a direct and intuitive manner (e.g., hiding explicit marshalling from the programmer).

Con: A native code interface gives up any of the type- and memory-safety guarantees of a language like Java or OCaml, allowing for defects that were previously avoided by construction.

Con: A native code interface complicates debugging as values pass from one language to another (debuggers typically only support one language).

Con: A native code interface requires the low-level interface programmer to understand data layout and representations in both languages.

Con: You could also mention any of the common bugs we covered in class, such as "failing to convert between language ints and native ints", "failing to properly handle null-embedded language strings in null-terminated native code", and "failing to properly manually interface native code with the language garbage collector".

7 Code Generation (10 points)

(a) *[5 pts]* Consider the following *incorrect* stack-machine code generation rule:

```
        cgen(if e1 = e2 then e3 else e4) =
                              cgen(e1)
                              push r1
                              cgen(e2)
                              pop t1
                              bneq r1 t1 false_branch
                              jmp true_branch
            false_branch:     cgen(e4)
            true_branch:      cgen(e3)
            end_if:
```

Write an expression e for which the above rule generates incorrect code. Indicate specifically what should happen when your expression e is evaluated as well as what mistakenly happens when the above rule is used to generate code.

The bug is that execution "falls through" from the false branch to the true branch.

if $1 = 0$ then out_int(1) else out_int(2)

Outputs 21 but should output 1.

(b) *[5 pts]* Consider the following Cool class declarations:

```
Class A {                              class B inherits A {
  p : Int;                                   e : Int ;
  t : String;                                z : Int ;
  w : Int;                                   y() : String { ... } ;
  x() : String { ... } ;                     x() : String { ... } ;
  z() : String { ... } ;                 }
}
```

Complete the following table describing the object (field) layouts:

	0 – 2	3	4	5	6	7
A	(header)	p	t	w		
B	(header)	p	t	w	e	z

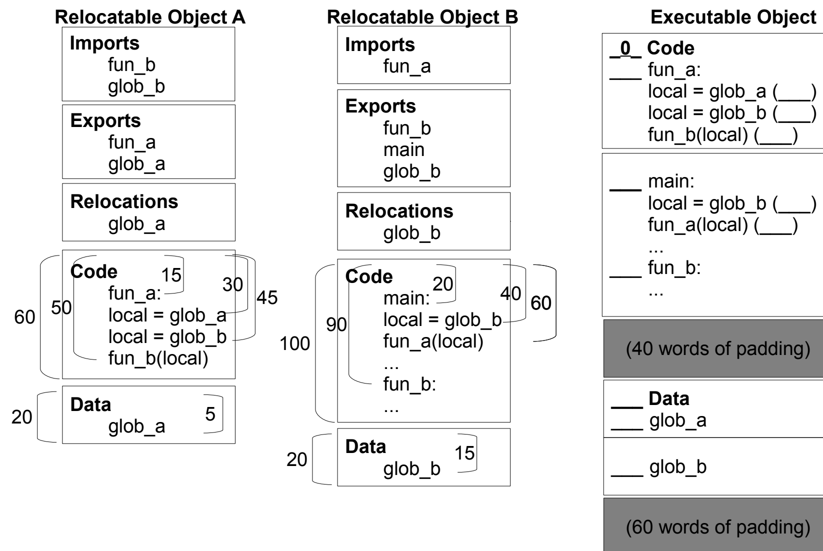
As well as the following table describing the dispatch table (vtable) layout:

	0	1	2	3
A	A's x()	A's z()		
B	B's (x)	A's z()	B's y()	

8 Linking (5 points)

(a) *[5 pts]* In the following scenario, two relocatable objects are being linked together to form an executable. Each object has an import table, an export table, a relocation table, a code segment and a data segment. The rounded braces indicate offsets within segments. For example, the use of `glob_a` within Object A's code segment occurs 30 words in, and that segment has a total length of 60. Similarly, the variable `glob_b` is located 15 words into Object B's data segment, and that segment has total size 20.

The linked executable is shown on the right. The page size is 100 words, causing the indicated amount of padding (greyed areas). Fill in the ten blanks with *absolute* post-link addresses. The linked code segment starts at absolute address 0. For example, you should fill in the `(___)` to the right of `glob_b` with the final linked address of `glob_b`.




```
0   Code
15  fun_a:
    local = glob_a  (205)
    local = glob_b  (235)
    fun_b(local)    (150)
80  main:
    local = glob_b  (235)
    fun_a(local)    (15)
150 fun_b:
200 Data
205 glob_a
235 glob_b
```

9 Game Theory (5 points)

(a) *[5 pts]* Consider the following *Nim* scenario. It is your turn. Indicate a winning move (e.g., write it out textually or circle the items you would take from a single heap). In the game board below, heap **A** has three items, heap **B** has four items, etc.

In binary: 3 xor 4 xor 5 xor 6 xor 2 = 6

Answer 1: Take all 6 from heap D.

Answer 2: Take 2 from heap C.

Answer 3: Take 2 from heap B.

And so on ...

10 Extra Credit (0 points)

"No Answer" is *not* valid on extra credit questions.

(a) *[1 pt]* Answer the following true-false questions about `SELF_TYPE`.

i. FALSE: `SELF_TYPE` is a dynamic type.

ii. FALSE: `SELF_TYPE` helps us to reject incorrect programs that are not rejected by the normal type system.

iii. FALSE: $T \leq \texttt{SELF_TYPE}_T$

iv. FALSE: A formal parameter to a method can have type `SELF_TYPE`.

v. TRUE: If the return type of method f is `SELF_TYPE` then the static type of $e_0.f(e_1, \ldots, e_n)$ is the static type of e_0.

(b) *[2 pts]* Cultural literacy. Below are the English translations or names for ten concepts or figures in world folklore, legend, religion or mythology. Each concept is associated with one of the ten most common languages (by current number of native-language speakers; Ethnologue estimate). For each concept, give the associated language. Be specific.

Bengali	Bankubabur Bandhu
Spanish	Don Juan
Portuguese	Endovelicus
Russian	Ilya Muromets
Arabic	Jinn
Hindi	Kamayani
Japanese	Kami
English	King Arthur
Chinese/Mandarin	The Eight Immortals
German	The Pied Piper