EECS 490 Final Exam, Fall 2016

Time: 100 minutes

- This exam is closed book, closed notebook.
- You may not provide your own scratch paper, but the last page is intentionally left blank for scratch work. Do **not** detach it from the rest of the exam, as that will interfere with the scanning process.
- Laptops, cell phones, calculators, and smartwatches are not allowed.
- Cell phones must be turned off.
- You are allowed one 8.5x11" sheet of notes as a reference.
- Any deviation from these rules will constitute an Honor Code violation.

- The exam consists of 5 questions, each with multiple parts.
- For Question 5, choose one of the two options to answer; we will only grade one answer.
- For multiple-choice and true/false questions, indicate your choices clearly.
- For short-answer and coding questions, write your answers clearly in the space provided.
- Do **not** write in the margins of a page (i.e. within an inch of the border), as anything written there will not show up when your exam is scanned.
- Assume all Python code is Python 3.5, all C++ code is C++11, all Scheme code is R5RS, all Java code is Java 8, and all Prolog code is SWI-Prolog 7. Adhere to these standards in the code you write as well.
- Your exam should have 10 pages, including this page and the blank page at the end.

Include your signature to indicate that you acknowledge the Honor Pledge, printed below.

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Name Solution _____

Uniqname _____

UMID _____

Signature _____

Name of person to your left _____

Name of person to your right _____

Question 1 (20 points)

Select the correct answers for each problem below from choices A-E. A problem may have multiple correct answers, so circle every answer that is correct.

a) Which of the following are examples of control-flow analysis that a compiler may perform? Circle all that apply:

A. Checking that a variable is initialized before use in all possible control paths.
B. Determining the types of compound expressions.
C. Determining if a function with a non-void return type will always reach a return statement.
D. Checking if the program contains unreachable or "dead" code.
E. Determining whether the program is free of infinite loops.

b) Which of the following uC code fragments contain *compile-time* errors? Circle all that apply:

A.
```
void main(string[] args)(boolean b) {
  println(args[b]);
}
```

B.
```
int add(int x)() {
  return x = x + 1;
}
```

C.
```
void main(string[] args)(int i, int j) {
  i = 1 / 2;
  j = 1 * 2;
  new foo(j / i);
}

struct foo(int x);
```

D.
```
struct bar(int x, int y);

void main(string[] args)(bar b) {
  b = new bar(3L, 3.14);
}
```

E.
```
void main(string[] args)() {
  println(args.length);
}
```

c) Given the C++ code below, what prints when the code is run? Circle only one answer.

```
#include <iostream>

#define INCREMENT(x, n) for (int i = 0; i < n; ++i) { ++x; }

int main() {
  int i = 0, j = 0;
  INCREMENT(i, 5);
  INCREMENT(j, 6);
  std::cout << i << "," << j << std::endl;
}
```

A. `0,0`
B. `0,6`
C. `5,6`
D. `5,0`
E. `1,6`

d) Given the following Prolog axioms, which of the Prolog queries below will be true? Circle all that apply.

```
is_tree(nil).
is_tree(tree(_, L, R)) :- is_tree(L), is_tree(R).
```

A. `?- is_tree(tree(1, nil, nil)).`
B. `?- is_tree(tree(1, tree(2, 3, nil))).`
C. `?- is_tree(tree(A, nil, nil)).`
D. `?- is_tree(tree(nil, tree(1, nil, nil), tree(3, nil, nil))).`
E. `?- is_tree(tree(1, tree(nil, nil), tree(4, 5, nil))).`

Question 2 (24 points)

For each statement below, circle **True** or **False**, or leave it blank. Unanswered statements will receive 50% credit.

a) **True** / **False** The general problem of determining whether a program is correct at compile time is undecidable.

b) **True** / **False** uC is a statically typed language.

c) **True** / **False** In operational semantics, the state σ keeps track of the result of evaluating every expression in the program.

d) **True** / **False** Inheritance and subtype polymorphism are fundamental features of object-oriented programming.

e) **True** / **False** By default, the attributes of a Python object are stored in a dictionary.

f) **True** / **False** In C++, every class type is descended from the root `object` type.

g) **True** / **False** In C++, it is impossible to access a public or protected member inherited from a base class if it is hidden by a member defined in the derived class.

h) **True** / **False** Java allows explicit constraints to be placed on parameters to a generic class or method.

i) **True** / **False** In C and C++, the `static` keyword makes a global variable or function visible outside its translation unit.

j) **True** / **False** In Prolog, a variable can unify with anything, including a compound term.

k) **True** / **False** In Prolog, a variable must be instantiated with a numerical value before an arithmetic operation can be applied to it.

l) **True** / **False** In Scheme, it is impossible to define a macro that swaps the values of two variables.

m) **True** / **False** Any computation that can be done in Scheme can also be done in template metaprogramming.

n) **True** / **False** The `decltype` specifier can be used to ensure a substitution failure in a C++ function template.

o) **True** / **False** It is a race condition if two threads concurrently read from the same memory location.

p) **True** / **False** Strategies for avoiding race conditions include message passing, locks, barriers, and synchronized data structures.

Question 3 (24 points)

a) Recall the simple language from the lecture on operational semantics:

$$
\begin{aligned}
P &\rightarrow S \\
S &\rightarrow \textbf{skip} \mid S;\ S \mid V\ =\ A \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end} \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end} \\
A &\rightarrow N \mid V \mid (\ A\ +\ A\ ) \mid (\ A\ -\ A\ ) \mid (\ A\ *\ A\ ) \\
B &\rightarrow \textbf{true} \mid \textbf{false} \mid (\ A\ <=\ A\ ) \mid (\ B\ \textbf{and}\ B\ ) \mid \textbf{not}\ B \\
V &\rightarrow \mathit{Identifier} \\
N &\rightarrow \mathit{Integer}
\end{aligned}
$$

Suppose we wanted to add a new statement to repeat an action a fixed number of times N:

$$S \rightarrow \textbf{repeat}\ N\ \textbf{do}\ S\ \textbf{end}$$

Write rules for the execution of the `repeat` statement in big-step operational semantics.

Fill in the recursive rule below:

$$\frac{\langle S,\sigma\rangle \rightarrow \sigma_1 \quad \langle \textbf{repeat}\ m\ \textbf{do}\ s\ \textbf{end},\sigma_1\rangle \rightarrow \sigma_2}{\langle \textbf{repeat}\ n\ \textbf{do}\ s\ \textbf{end},\sigma\rangle \rightarrow \sigma_2} \quad \text{if } n>0, \text{ where } m=n-1$$

Also fill in the rule for the base case:

$$\frac{}{\langle \textbf{repeat}\ 0\ \textbf{do}\ s\ \textbf{end},\sigma\rangle \rightarrow \sigma}$$

b) The following is part of the definition of a vector type in Java:

```
 1 public class Vector<T> {
 2   private T[] elements;
 3   public Vector() {
 4     elements = new T[10];
 5   }
 6   public T get(int index) {
 7     return elements[index];
 8   }
 9   ...
10 }
```

Attempting to compile this code results in the following error:

```
> javac Vector.java
Vector.java:4: error: generic array creation
    elements = new T[10];
               ^
1 error
```

Modify the code to fix the error. Write only the lines that need be modified, preceded by the line number, as in:

```
3 private Vector() {
```

```
4  elements = (T[]) new Object[10];
```

-- OR --

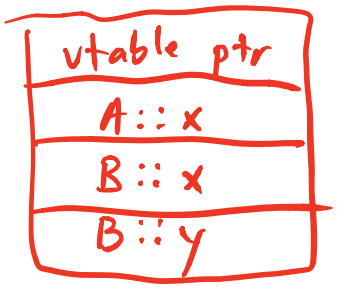
```
2  private Object[] elements;
4  elements = new Object[10];
7  return (T) elements[index];
```

c) Consider the following C++ code:

```
struct A {
  int x = 0;
  void foo() {
    cout << "A::foo()" << endl;
  }
  virtual void bar() {
    cout << "A::bar()" << endl;
  }
};

struct B : A {
  int x = 1;
  int y = 2;
  virtual void baz() {
    cout << "B::baz()" << endl;
  }
  void foo() {
    cout << "B::foo()" << endl;
  }
  virtual void bar() {
    cout << "B::bar()" << endl;
  }
};
```

i. Draw a picture illustrating the contents an object of type `B`. Clearly indicate what each entry in the object is, and if it is a member, which type contains its definition (e.g. `T::member`).



ii. Draw the layout of the vtable for type `B`. Clearly indicate the name of each member as well as which type contains its definition.

iii. What does each line in the following code print?

```
A *ptr = new B;
cout << ptr->x << endl;   // prints: 0

ptr->foo();               // prints: A::foo()

ptr->bar();               // prints: B::bar()
```

Question 4 (17 points)

a) Recall that a Church numeral N is represented as a function that takes in another function f and returns a function that applies f to its argument N times. In Scheme, the following are the Church numerals:

```
(lambda (f) (lambda (x) x))              ; 0
(lambda (f) (lambda (x) (f x)))          ; 1
(lambda (f) (lambda (x) (f (f x))))      ; 2
(lambda (f) (lambda (x) (f (f (f x)))))  ; 3
```

i. First, fill in the `apply_string()` function in Python to return a string representing the application of `f` to `x` `n` times.

```
def apply_string(n):
    """Returns a Scheme expression that applies f to x a total of
    n times.
    >>> apply_string(0)
    'x'
    >>> apply_string(1)
    '(f x)'
    >>> apply_string(2)
    '(f (f x))'
    """
    if n == 0:
        return 'x'
    return '(f {})'.format(apply_string(n-1))
```

--OR--

```
return "(f " * n + "x" + ")" * n
```

--OR--

```
res = 'x'
for i in range(n):
    res = '(f {})'.format(res)
return res
```

ii. Now fill in the `church()` function to return the Scheme representation of a Church numeral. Use `apply_string()` in your solution.

```
def church(n):
    """Returns a Scheme representation of the Church numeral for n.
    >>> church(0)
    '(lambda (f) (lambda (x) x))'
    >>> church(1)
    '(lambda (f) (lambda (x) (f x)))'
    >>> church(2)
    '(lambda (f) (lambda (x) (f (f x))))'
    """
    skeleton = '(lambda (f) (lambda (x) {}))'
    return skeleton.format(apply_string(n))
```

b) In Python, a string can be multiplied by a non-negative integer `N`, which evaluates to a new string with the original string repeated `N` times:

```
>>> 'z' * 3
'zzz'
```

Write a set of overloads for a `mult()` function in C++ that performs this string repetition when the first argument is a string and the second a non-negative integer. On the other hand, if the two arguments both have numerical type, then `mult()` multiplies the two arguments and returns the result. (Hint: Only two overloads are necessary.)

Example:

```
cout << mult(3, 4.1) << endl;
cout << mult("abc", 3) << endl;
```

This prints:

```
12.3
abcabcabc
```

```
template<class A, class B>
auto mult(A a, B b) -> decltype(a * b) {
    return a * b;
}

string mult(string a, int b) {
    string res;
    for (int i=0; i<b; i++) {
        res += a;
    }
    return res;
}
```

Question 5 (15 points)

For this question, choose one of the problems below to implement. We will only grade your solution to one of the questions. If you write code for both, you must clearly indicate which one you want us to grade, or we will choose one arbitrarily. You must write your code in SWI-Prolog 7. You may use the built-in arithmetic and list operations. You may **not** use any built-in predicates. You may write any helper predicates you want. (Reminder: the `!` operator in the queries is the cut operator in Prolog, which terminates the search after the first solution is found.)

a) Write a Prolog predicate `filter_nonempty` that relates a list of lists to another that contains only the nonempty lists from the first, preserving order. Example:

```
?- filter_nonempty([[1], [], [2, 3, 4], [], [a, b]], X), !.
X = [[1], [2, 3, 4], [a, b]]
```

b) Write a Prolog predicate `any_contains` that relates a list of lists and an item, and is true if and only if at least one of the inner lists contains the item. Examples:

```
?- any_contains([[1], [2, 3], [4]], a), !.
false.
?- any_contains([[1], [2, 3], [4]], 3), !.
true.
```

Write your code below

a)
```
filter_nonempty([], []).
filter_nonempty([[]|Bs], Cs) :-
  filter_nonempty(Bs, Cs).
filter_nonempty([[A|As]|Bs], [[A|As]|Cs]) :-
  filter_nonempty(Bs, Cs).
-- OR (for last rule) --
filter_nonempty([A|As], [A|Cs]) :-
  filter_nonempty(As, Cs).
```

b)
```
contains([X|_], X).
contains([_|As], X) :- contains(As, X).
any_contains([L|_], X) :- contains(L, X).
any_contains([_|Ls], X) :- any_contains(Ls, X).
```

This page intentionally left blank. We will not grade any work here. Do NOT detach from exam.