EECS 490 Final Exam, Fall 2017

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. Please do **not** detach it.
- 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 multiple-choice and true/false questions, indicate your choice by filling in the appropriate square or bubble. Fill in the square or bubble completeley, and make sure to erase completely if you change your answer.
- 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 3.6, all C++ code is C++14, and all Prolog code is SWI-Prolog 7. Adhere to these standards in the code you write as well.
- Your exam should have 12 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 _____

Uniqname _____

UMID _____

Signature _____

Name of person to your left _____

Name of person to your right _____

Question 1 (15 points)

a) Which of the following uC code fragments contain *compile-time* errors? Fill in the square above each erroneous fragment:

A. ☐

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

B. ☐

```
int get_value()(int x) {
  return x;
}
```

C. ☐

```
void main(string[] args)() {
  args[0] = null;
}
```

D. ☐

```
void main(string[] args)() {
  args >> null;
}
```

E. ☐

```
void main(string[] args)() {
  println(new foo().s);
}

struct foo(string s);
```

b) Consider the following Prolog program:

```
shorter_than(List, Size) :-
    length(List, Length),
    Length < Size.

shorter(List1, List2) :-
    length(List2, Length2),
    shorter_than(List1, Length2).
```

For each of the following queries, determine if the query succeeds, fails, or results in a runtime error:

Query	**True**	**False**	**Error**
`shorter_than([1, 2], 3).`	○	○	○
`shorter_than([1, 2], L).`	○	○	○
`shorter([1, 2], [3]).`	○	○	○
`shorter([1, 2], [_, _]).`	○	○	○
`shorter([1, 2], [A, B, C]).`	○	○	○

c) Consider the following C++ code:

```
#include <vector>

template<typename T>
struct A {
  using type = int;
  static const bool value = true;
};

template<typename T>
struct A<std::vector<T>> {
  static const bool value = false;
};

template<typename T>
bool func(T a, typename A<T>::type b = 0) {
  return true;
}
```

For each of the following expressions, determine if the value is true or false, or if the expression results in a compile-time error:

Expression	**True**	**False**	**Error**
`A::value`	◯	◯	◯
`A<int>::value`	◯	◯	◯
`A<std::vector<int>>::value`	◯	◯	◯
`func(3)`	◯	◯	◯
`func(std::vector<int>())`	◯	◯	◯

Question 2 (20 points)

For each statement below, fill in the bubble to the left of **True** or **False**, or leave them both blank. Unanswered statements will receive 50% credit.

a) ◯**True** / ◯**False** In Java, if a class `Counter` implements `Comparable<Counter>`, then two `Counter` objects can be compared using the < operator.

b) ◯**True** / ◯**False** In Python 3, every class that is not `object` derives from the root `object` class.

c) ◯**True** / ◯**False** In big-step operational semantics, the result of a transition can only be a value, state, or one of each.

d) ◯**True** / ◯**False** In a formal type rule, the type context Γ maps an **expression** to its type.

e) ◯**True** / ◯**False** In C++, a derived-class method does not override a virtual base-class method if the parameter lists of the two methods differ.

f) ◯**True** / ◯**False** In C++, it is permissible for two entities to share the same name if they belong to different namespaces.

g) ◯**True** / ◯**False** In C++, in order for the expression `static_cast<A *>(new B)` to compile, `A` and `B` must each define at least one virtual method.

h) ◯**True** / ◯**False** In Prolog, an uninstantiated variable does not unify with a variable that is instantiated.

i) ◯**True** / ◯**False** In Scheme, it is possible for names defined within a macro body to conflict with names that are in scope when the macro is used.

j) ◯**True** / ◯**False** Incorrect use of locks can result in deadlock, where two threads are each waiting on a resource held by the other.

Question 3 (25 points)

a) Recall the following language from the lecture on formal type systems:

$$
\begin{aligned}
P &\rightarrow E \\
E &\rightarrow N \\
&\quad \mid B \\
&\quad \mid (\ E\ +\ E\ ) \\
&\quad \mid (\ E\ -\ E\ ) \\
&\quad \mid (\ E\ *\ E\ ) \\
&\quad \mid (\ E\ <=\ E\ ) \\
&\quad \mid (\ E\ \textbf{and}\ E\ ) \\
&\quad \mid \textbf{not}\ E \\
&\quad \mid (\ \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E\ ) \\
&\quad \mid (\ \textbf{let}\ V\ =\ E\ \textbf{in}\ E\ ) \\
&\quad \mid V \\
V &\rightarrow Identifier \\
N &\rightarrow IntegerLiteral \\
B &\rightarrow \textbf{true} \\
&\quad \mid \textbf{false}
\end{aligned}
$$

Suppose we wanted to expand our language to include pairs, denoted by an expression such as $\{1,2\}$ or $\{3,\textbf{false}\}$, as well as **car** and **cdr** operators to obtain the first and second out of a pair:

$$
\begin{aligned}
E &\rightarrow \{\ E\ ,\ E\ \} \\
&\quad \mid \textbf{car}\ E \\
&\quad \mid \textbf{cdr}\ E
\end{aligned}
$$

Let $T_1 \times T_2$ denote the type of a pair where the first element has type T_1 and the type of the second is T_2. For example, $\{3,\textbf{false}\}$ has type $Int \times Bool$.

i. Fill in the formal type rule for computing the type of a pair expression:

$$\frac{}{\Gamma \vdash \{t_1,\ t_2\}\ :}$$

ii. Now fill in the rule for applying **car** to a pair:

$$\frac{}{\Gamma \vdash \textbf{car}\ t_1\ :}$$

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

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

Given an initial state σ, where $\sigma(x) = 3$, determine the full derivation tree for the code fragment below using the big-step operational-semantics rules for the language.

$$\textbf{if}\ (x <= 3)\ \textbf{then}\ x = 4\ \textbf{else skip end}$$

The following are some relevant operational-semantics rules:

$$\frac{\langle a_1, \sigma\rangle \rightarrow n_1 \quad \langle a_2, \sigma\rangle \rightarrow n_2}{\langle (a_1 <= a_2), \sigma\rangle \rightarrow \textbf{true}} \quad \text{if } n_1 \leq n_2 \qquad \frac{\langle a_1, \sigma\rangle \rightarrow n_1 \quad \langle a_2, \sigma\rangle \rightarrow n_2}{\langle (a_1 <= a_2), \sigma\rangle \rightarrow \textbf{false}} \quad \text{if } n_1 > n_2$$

$$\frac{\langle b, \sigma\rangle \rightarrow \textbf{true} \quad \langle s_1, \sigma\rangle \rightarrow \sigma_1}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end}, \sigma\rangle \rightarrow \sigma_1} \qquad \frac{\langle b, \sigma\rangle \rightarrow \textbf{false} \quad \langle s_2, \sigma\rangle \rightarrow \sigma_2}{\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end}, \sigma\rangle \rightarrow \sigma_2} \qquad \frac{\langle a, \sigma\rangle \rightarrow n}{\langle v = a, \sigma\rangle \rightarrow \sigma[v := n]}$$

Your derivation below:

c) Consider the following C++ code:

```
#include <iostream>
using std::cout;
using std::endl;

struct A {
  int x;
  virtual void foo() { cout << "A::foo()" << endl; }
  void baz() { cout << "A::baz()" << endl; }
};

struct B : A {
  virtual void bar() { cout << "B::bar()" << endl; }
  virtual void foo() { cout << "B::foo()" << endl; }
  void baz() { cout << "B::baz()" << endl; }
};

struct C : B {
  int z;
  virtual void baz() { cout << "C::baz()" << endl; }
};
```

i. Draw a picture illustrating the contents of an object of type `C`. 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 `C`. 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?

```
C c;
A* ptr = &c;

c.baz();                // prints: _____

ptr->baz();             // prints: _____

c.foo();                // prints: _____

ptr->foo();             // prints: _____
```

Question 4 (20 points)

a) Suppose we modified uC by adding the ability to customize the comparison operators on user-defined types. A type `foo` would support the comparison operators if a uC function with the following signature is defined:

```
int foo_compare(foo lhs, foo rhs)() { ... }
```

This function should return a negative value if `lhs` is less than `rhs`, zero if they are equal, or a positive value if `lhs` is greater than `rhs`.

The following is an example of using a comparison operator:

```
foo foo_min(foo a, foo b)() {
  if (a < b) { return a; }
  else { return b; }
}
```

Fill in the Python function `gen_comparison_op()` below. Given the name of a type (e.g. `foo`) and the string representation of a comparison operator (e.g. <=), the function prints out C++ source code for the given overloaded comparison operator. The overloaded operator should be defined as a standalone C++ function, and it should call the uC comparison function for the type (e.g. `foo_compare()`). You may use the following macros from Project 5:

Macro	Purpose
`_UC_TYPEDEF(name)`	A raw user-defined type of the given name, used in type definitions
`_UC_TYPE(name)`	A user-defined type wrapped in a reference
`_UC_FUNCTION(name)`	A function of the given name

```
def gen_comparison_op(typename, op):
    """Generates a C++ overloaded comparison operator for the
    given type and operator, printing it to standard output.
    typename must be a string representing the uC name of the
    type. op must be one of the strings '<', '<=', '==', '!=',
    '>', '>='.
    Example: gen_comparison_op('foo', '<=')
             Prints a C++ operator overload that calls the uC
             foo_compare() function.
    """
    # your code below
```

b) In the definition of `Counter` below, write overloads of `add()` such that it can be called on both a value of type `T` as well as on another `Counter` object, as in the following:

```
int main() {
  Counter<int> c1;     // counter that holds an int, count is 0
  Counter<double> c2; // counter that holds a double, count is 0
  c1.add(3);           // c1's count is now 3
  c2.add(-4);          // c2's count is now -4
  c2.add(-2.2);        // c2's count is now -6.2
  c1.add(c2);          // add c2's count to c1; c1 now -3
  c2.add(c1);          // add c1's count to c2; c2 now -9.2
  std::cout << c2.get() << std::endl; // prints -9.2
}
```

Complete the definition of `Counter`:

```
template<typename T>
class Counter {
  T count = {}; // value initialize the count
public:
  T get() const {
    return count;
  }

  // your code here









};
```

Question 5 (20 points)

For this question, you must write your code in SWI-Prolog 7. You may use the built-in unification, negation, and list operations. You may **not** use any built-in predicates. You may write any helper predicates you want. Your predicates do **not** have to be "tail recursive". Your code must **not** have singleton variables that are not anonymous.

a) Write a Prolog predicate `remove` that relates an item, a list, and another list that is the same as the first one but with all occurrences of the item removed. Examples:

```
?- remove(3, [3, 1, 3, 2, 3], X).
X = [1, 2] .
?- remove(4, [3, 1, 3, 2, 3], X).
X = [3, 1, 3, 2, 3] .
```

Your code must not produce incorrect solutions if the user asks for more solutions. Recall that the `findall` predicate finds all solutions to a query. Then `remove` should be defined such that:

```
?- findall(X, remove(3, [3, 1, 3, 2, 3], X), Solutions).
Solutions = [[1, 2]].
```

Hint: You will need to use negation in your solution.

```
% remove(Item, List, Result)
%   True if removing all occurrences of Item in List results in
%   Result.
% your code below
```

b) Now write a predicate `remove_all` that relates a list of items, a second list, and a third list that is the same as the second but with all occurrences of the items in the first list removed. Example:

```
?- remove_all([3, 1], [3, 1, 3, 2, 3], X).
X = [2] .
```

You **MUST** use `remove` in your solution, and you may assume it is correct. You will not need negation in your code for `remove_all`.

```
% remove_all(Items, List, Result)
%   True if removing all occurrences of the elements in Items from
%   List results in Result.
% your code below
```

This page intentionally left blank. We will not grade any work here.