

# 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 completely, 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 Solution

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;
}
```

} We discarded this, since it depends on whether the optional part of Project 4 is implemented.

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).	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
shorter_than([1, 2], L).	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
shorter([1, 2], [3]).	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
shorter([1, 2], [_ , _]).	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
shorter([1, 2], [A, B, C]).	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>

L is uninstantiated, so it cannot be compared with <.

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	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
A<int>::value	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
A<std::vector<int>>::value	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
func(3)	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
func(std::vector<int>())	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

*A is an uninstantiated template.*

*Fails to substitute since A<std::vector<int>>::type does not exist.*

## 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  $\Gamma$  maps an expression to its type. *It maps variables to their types.*
- 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:

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

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

$$\begin{aligned}
 E &\rightarrow \{E, E\} \\
 &| \text{car } E \\
 &| \text{cdr } E
 \end{aligned}$$

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,  $\{3, \text{false}\}$  has type  $\text{Int} \times \text{Bool}$ .

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

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2}$$

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

$$\frac{\Gamma \vdash t_1 : T_1 \times T_2}{\Gamma \vdash \text{car } t_1 : T_1}$$

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

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

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

**if  $(x \leq 3)$  then  $x = 4$  else skip end**

The following are some relevant operational-semantics rules:

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

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

Your derivation below:

$$\begin{array}{c}
 \overline{\langle x, \sigma \rangle \rightarrow 3} \quad \overline{\langle 3, \sigma \rangle \rightarrow 3} \\
 \hline
 \langle (x \leq 3), \sigma \rangle \rightarrow \mathbf{true}
 \end{array}
 \qquad
 \begin{array}{c}
 \overline{\langle 4, \sigma \rangle \rightarrow 4} \\
 \hline
 \langle x = 4, \sigma \rangle \rightarrow \sigma[x := 4]
 \end{array}$$


---


$$\langle \mathbf{if } (x \leq 3) \mathbf{ then } x = 4 \mathbf{ else skip end}, \sigma \rangle \rightarrow \sigma[x := 4]$$

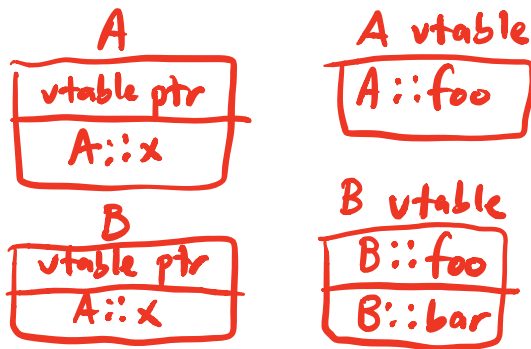
c) Consider the following C++ code:

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

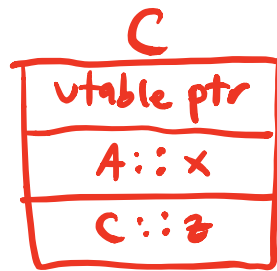
struct A {
    int x;
    virtual void foo() { cout << "A::foo()" << endl; }
    void baz() { cout << "A::baz()" << endl; }
};

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

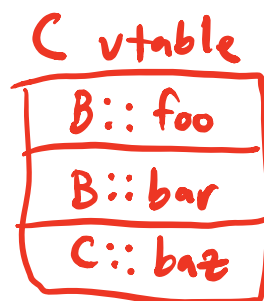
struct C : B {
    int z;
    virtual void baz() { cout << "C::baz()" << endl; }
};
```



- 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:
```

**C::baz()**

```
ptr->baz();
```

```
// prints:
```

**A::baz()**

```
c.foo();
```

```
// prints:
```

**B::foo()**

```
ptr->foo();
```

```
// prints:
```

**B::foo()**

## 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
<code>_UC_TYPEDEF (name)</code>	A raw user-defined type of the given name, used in type definitions
<code>_UC_TYPE (name)</code>	A user-defined type wrapped in a reference
<code>_UC_FUNCTION (name)</code>	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
```

```
code = ('bool operator {0} (-UC-TYPE({1}) a,\n    '\n    -UC-TYPE({1}) b) {{\n    '\n    return -UC-FUNCTION({1}_compare)\n    '\n    (a, b) {0} 0;\n    '\n    }}\n    '\n    )  
print (code.format(op, typename))
```



- 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

```
void add(const T &value) {
    count += value;
}
```

```
template<typename U>
void add(const Counter<U> &other) {
    count += other.get();
}
```

```
// parameters must be pass by value or by
// reference to const
// second overload must be a function template
// must call get(), since Counter<T> does not
};
// have access to the private members of
// Counter<U> if T ≠ U
```

## 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
```

```
remove(-, [], []).  
remove(Item, [Item | Rest], Result) :-  
    remove(Item, Rest, Result).  
remove(Item, [Other | Rest], [Other, Result]) :-  
    Item \= Other,  
    remove(Item, Rest, Result).
```

- 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
```

```
remove_all([], List, List) .
```

```
remove_all([First|Rest], List, Result) :-  
    remove(First, List, Removed),  
    remove_all(Rest, Removed, Result) .
```

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