Final Exam

CS131: Programming Languages

Tuesday, March 18, 2014

1.	5.	9.
2.	6.	10.
3.	7.	
4.	8.	

Name:_____

ID:_____

Rules of the game:

- **Write your name and ID number above.**
- The exam is closed-book and closed-notes.
- Please write your answers directly on the exam. Do not turn in anything else.
- The exam ends promptly at 11am.
- Read questions carefully. Understand a question before you start writing. *Note: Some multiple-choice questions ask for a single answer, while others ask for all appropriate answers.*
- Relax!

1. (5 points each) This question explores how an OCaml datatype can be used to represent the nonnegative integers (sometimes called the *natural numbers*). Consider the following datatype:

   ```
   type nat = Z | S of nat
   ```

   In this representation of nonnegative integers `Z` represents the number 0 (the constructor `Z` stands for "zero") and `S n` represents the integer that is one bigger than the number represented by `n` (the constructor `S` stands for "successor"). For example, the number 3 would be represented by the value `S(S(S Z))`.

   (a) Implement a function `plus` of type `nat -> nat -> nat` which takes two integers in the above representation and returns a `nat` value representing their sum.

   Two possible solutions:

   ```
   let plus m n =
       match m with
          Z -> n
       | S m' -> plus m' (S n)

   let plus m n =
       match m with
          Z -> n
       | S m' -> S (plus m' n)
   ```

   (b) Implement a function `leq` of type `nat -> nat -> bool` which takes two integers `m` and `n` in the above representation and returns a boolean value indicating whether `m` is less than or equal to `n`.

   ```
   let rec leq m n =
       match (m,n) with
         (Z, _) -> true
       | (_, Z) -> false
       | (S m', S n') -> leq m' n'
   ```

2. (2 points each) Consider the following Java interfaces and classes. You may assume that each class has a zero-argument constructor.

```
interface Greeter {
  void greet();
}
class Person implements Greeter {
  public void greet() { this.hello(new Integer(3)); }
  public void hello(Object o) { System.out.println("hello object"); }
}
class CSPerson extends Person {
  public void hello(Object o) { System.out.println("hello world!"); }
}
class FrenchPerson extends Person {
  public void hello(Object o) { System.out.println("bonjour object"); }
  public void hello(String s) { System.out.println("bonjour " + s); }
}
```

For each code snippet below (which is independent of all other snippets), either indicate that the snippet causes a compile-time error, causes a run-time error, or show what Java would print as a result of evaluating that snippet.

(a)
```
Person p = new CSPerson();
p.greet();
```
hello world!

(b)
```
CSPerson p = new CSPerson();
p.greet();
```
hello world!

(c)
```
Greeter g = new FrenchPerson();
g.hello("joe");
```
compile-time error

(d)
```
Person p = new FrenchPerson();
p.hello("joe");
```
bonjour object

3. (2 points each) Consider a subset of the types from the previous problem:

```
class Person {
  public void greet() { this.hello(new Integer(3)); }
  public void hello(Object o) { System.out.println("hello object"); }
}
class CSPerson extends Person {
  public void hello(Object o) { System.out.println("hello world!"); }
}
```

And consider this snippet of code using those types:

```
CSPerson p = new CSPerson();
p.greet();
```

Circle the best answer for each question below.

(a) Suppose Java lacked support for inheritance, but everything else about the language was unchanged. So for example, `extends` declares a subtyping relationship between two classes but not an inheritance relationship. What would be the effect on the above code?

i. no change in behavior
ii. the `CSPerson` class would not pass the static typechecker
iii. the code snippet above would not pass the static typechecker
iv. the code snippet would cause a run-time error
v. the code snippet would print something different than it currently does

ii

(b) Suppose Java lacked support for dynamic dispatch, but everything else about the language was unchanged. What would be the effect on the above code?

i. no change in behavior
ii. the `CSPerson` class would not pass the static typechecker
iii. the code snippet above would not pass the static typechecker
iv. the code snippet would cause a run-time error
v. the code snippet would print something different than it currently does

v

(c) Suppose Java lacked support for static overloading, but everything else about the language was unchanged. What would be the effect on the above code?

i. no change in behavior
ii. the `CSPerson` class would not pass the static typechecker
iii. the code snippet above would not pass the static typechecker
iv. the code snippet would cause a run-time error
v. the code snippet would print something different than it currently does

i

4. (5 points) Consider the following Java interface, which represents booleans:

```
interface Bool {
  Bool logicalAnd(Bool b);
  void ifThenElse(CodeBlock b1, CodeBlock b2);
}
interface CodeBlock { void execute(); }
```

The `logicalAnd` method performs "and" for booleans (but unlike the built-in `&&` operator, `logicalAnd` is not short circuiting). The `ifThenElse` method acts like a traditional if-then-else statement, executing one of the two code blocks (represented by the interface `CodeBlock`) depending on the value of the boolean.

It turns out that we can implement this functionality without using the primitive booleans, or any primitive values, at all. Define classes `True` and `False`, each of which should implement the `Bool` interface.

For example, here is some sample client code:

```
Bool b1 = new False();
Bool b2 = new True();
Bool b3 = b1.logicalAnd(b2);
b3.ifThenElse(
  new CodeBlock() { public void execute() { System.out.println("hi!"); } },
  new CodeBlock() { public void execute() { System.out.println("bye!"); } }
);
```

Executing the above code will cause `bye!` to be printed to standard output.

Your solution may not use `instanceof` tests or any other ways of asking an object whether it represents true or false. Your solution also may not use any primitive data in Java, such as booleans, integers, and Strings, and it may not use any built-in control structures such as `if`, `for`, and `while`.

```
class True implements Bool {
    public Bool logicalAnd(Bool b) { return b; }
    public void ifThenElse(CodeBlock b1, CodeBlock b2) { b1.execute(); }
}

class False implements Bool {
    public Bool logicalAnd(Bool b) { return this; }
    public void ifThenElse(CodeBlock b1, CodeBlock b2) { b2.execute(); }
}
```

5. (5 points) Continuing the example from the previous question, we can use our notion of booleans and conditional statements to implement while loops. Here is an interface for the condition used to guard a loop:

```
interface LoopGuard {
  Bool evaluate();
}
```

The `evaluate` method evaluates the condition to produce a boolean result (using the `Bool` type from the previous page). Implement a method called `whileTrue` that acts like a traditional while loop, but using our new types. The method takes a `LoopGuard` and a `CodeBlock` and it repeatedly executes the code block while the loop guard is true.

Again, your solution may not use `instanceof` tests or any other ways of asking an object whether it represents true or false. Your solution also may not use any primitive data in Java, such as booleans, integers, and Strings, and it may not use any built-in control structures such as `if`, `for`, and `while`.

Fill in the code below with your solution. (Technically the two parameters need to be declared `final` if you refer to them within an anonymous class in your implementation, but we'll not worry about that.)

```
class Loops {
  void whileTrue(LoopGuard guard, CodeBlock body) {
     // YOUR CODE HERE




  }
}
```

```
class Loops {
  void whileTrue(final LoopGuard guard, final CodeBlock body) {
      Bool b = guard.evaluate();
      b.ifThenElse(
        new CodeBlock() {
          public void execute() {
            body.execute();
            new Loops().whileTrue(guard, body); } },
        new CodeBlock() { public void execute() {} });

  }
}
```

6. (2 points each) Consider the following class:

```
class MyInt {
  int i;
  MyInt(int k) { this.i = k; }
  void swap1(MyInt j) {
    int tmp = j.i;
    j = new MyInt(this.i);
    this.i = tmp;
  }
  void swap2(MyInt j) {
    MyInt tmp = j;
    j.i = this.i;
    this.i = tmp.i;
  }
  void swap3(int j) {
    int tmp = j;
    j = this.i;
    this.i = tmp;
  }
}
```

Answer each of the following questions with respect to the following definitions:

```
MyInt m1 = new MyInt(3);
MyInt m2 = new MyInt(4);
```

Each question is completely independent of all other questions.

(a) What are the values of `m1.i` and `m2.i` after a call `m1.swap1(m2)`? 4 and 4

(b) What are the values of `m1.i` and `m2.i` after a call `m1.swap2(m2)`? 3 and 3

(c) What are the values of `m1.i` and `m2.i` after a call `m1.swap3(m2.i)`? 4 and 4

(d) Java's parameter-passing style is best described as which one of the following?

i. call by value
ii. call by reference
iii. call by value for primitives (integers, floats, etc.); call by reference for objects
iv. call by reference for primitives (integers, floats, etc.); call by value for objects

i

7. (3 points each) **Circle all answers that apply.**

(a) Which of these are advantages of throwing an exception over simply returning a distinguished error value (e.g., `null`)?

i. Exceptions never cause the program execution to be terminated.

ii. Exceptions clearly distinguish erroneous from normal behavior.

iii. Exceptions implicitly propagate up the dynamic call chain to the nearest handler.

iv. Exceptions can contain additional information about the error that occurred.

ii, iii, and iv

(b) Consider a successfully typechecked Java method `void m() throws Exn1, Exn2 {...}`, where `Exn1`, `Exn2`, and `Exn3` are all direct subclasses of `Exception`. Which of the following are true?

i. At least one of `Exn1` and `Exn2` will be thrown whenever `m` is called.

ii. A call to `m` will only throw `Exn2` if `Exn1` has already been caught.

iii. If `Exn1` is thrown during the execution of `m` it will definitely not be caught before the method ends.

iv. If `Exn3` is thrown during the execution of `m` it will definitely be caught before the method ends.

iv

(c) Consider again the method `m` from the prior problem. Which of the following are true?

i. Java requires every caller of `m` to catch exceptions of type `Exn1` as well as `Exn2`.

ii. Java requires every caller of `m` to catch exceptions of type `Exn3`.

iii. Java requires every caller of `m` to either catch exceptions of type `Exn1`, or to include `Exn1` or a supertype of `Exn1` in its `throws` clause.

iv. Java ensures that every caller of `m` does not include `Exn3` in its `throws` clause.

iii

8. (2 points each) What will the Prolog interpreter print in response to each of these queries? **You should assume that Prolog is performing the "occurs check" as part of unification.**

   (a) `f(X,a) = f(b,Y)`
   {X=b, Y=a}

   (b) `f(W,W) = f(X,g(Z))`
   {W=g(Z), X=g(Z)}

   (c) `f(X,a) = f(b,X)`
   no

   (d) `f(X,Y) = f(Y,g(X))`
   no (occurs check failure)

   (e) `[a] = [X, Y]`
   no

   (f) `[a] = [X | Y]`
   {X=a, Y=[]}

   (g) `[Y, b] = [X, Y]`
   {X=b, Y=b}

   (h) `[Y, b] = [X | Y]`
   {X=[b], Y=[b]}

9. (5 points) Define a Prolog predicate `merge(X,Y,Z)` which is provable when the list `Z` is the result of merging the lists `X` and `Y`. You may assume that `X` and `Y` are given as constant lists whose elements are integers in sorted order from least to greatest, and that the two lists have no elements in common. `Z` must also then be a list of integers in sorted order from least to greatest. For example, the query `merge([1,3,5],[2,6],W)` should succeed with a single solution of `W` mapped to `[1,2,3,5,6]`, and its execution should terminate. *You may not use any built-in predicates on lists, such as* `sort` *and* `append`.

```
merge([], L, L).
merge(L, [], L).
merge([H1|T1], [H2|T2], [H1|T3]) :- H1 < H2, merge(T1, [H2|T2], T3).
merge([H1|T1], [H2|T2], [H2|T3]) :- H2 < H1, merge([H1|T1], T2, T3).
```

10. (3 points each) Consider a Prolog predicate `member(E, L)`, which is provable when `E` is an element of list `L`. We can define the `member` predicate as follows:

```
member(X, [H|T]) :- member(X, T).
member(H, [H|_]).
```

In the following questions, you need not show the proof tree (though building the tree will likely help you determine the answers).

(a) Consider the query `member(X, [1,2,3])`. Show the solutions to this query that the Prolog interpreter would produce, in the same order that they would be produced. If there are no solutions produced (either because there are no answers or because the interpreter goes into an infinite loop), write NONE. If there are an infinite number of solutions, show the first three solutions that would be produced and then write INFINITE.

`X=3`
`X=2`
`X=1`

(b) Consider the query `member(1,X)`. Show the solutions to this query that the Prolog interpreter would produce, in the same order that they would be produced. If there are no solutions produced (either because there are no answers or because the interpreter goes into an infinite loop), write NONE. If there are an infinite number of solutions, show the first three solutions that would be produced and then write INFINITE. NONE

(c) Suppose that the order of the two rules above in the definition of `member` is reversed. Given this new definition of `member`, answer the same question as in part (a) above.

`X=1`
`X=2`
`X=3`

(d) Suppose that the order of the two rules above in the definition of `member` is reversed. Given this new definition of `member`, answer the same question as in part (b) above.

`X=[1|_]`
`X=[_,1|_]`
`X=[_,_,1|_]`
`INFINITE`