Computer Science 365
Final exam review

Retrospective of the course. Looking back on this semester, we can see the following themes to have emerged. We hope that these ideas have been made clear and have illuminated your understanding of programming languages, which are the software developer's most important tools.

- We have formal tools to capture the **specification of a programming language**, from the grammar to the type rules to how the program executes.
- There are both **static and dynamic** aspects of a programming language (and a program), and with the specification, we can write tools to analyze static aspects of a program; the most important static aspect is typing.
- Using the specification, we can **reason formally about programming languages**, such as proving that if a program holds to certain rules, then the program has some aspect of correctness.
- The specification also leads to the **implementation of programming languages**, and in particular, there are three ways to implement a language: **interpretation**, **compilation**, and **source-to-source compilation**.
- Both interpretation and source-to-source compilation prove the **equivalence among programming languages**; in particular, writing an interpreter for an object-oriented language in a functional language and vice versa shows the equivalence between OO and functional programming; writing a source-to-source compiler from a big language to a small language shows that many common features do not add expressive power to a language, just convenience.

The goal of the final exam is to strengthen and evaluate your understanding of these concepts. The following topics for review are largely presented as potential final exam questions. Questions on the final will tend to be variations on some of these questions.

A. Fundamental vocabulary. These are terms which you learned in your first programming language, but which you now must have a precise understanding of: Value, expression, statement, variable, declaration, type, scope. Static vs. dynamic; that is, what things are known at compile time, and what things are not known until runtime.

B. Object-oriented programming.

1. What characterizes object-oriented programming? (Packaging data and functionality together. Do not say "classes" or "polymorphism" since we have seen that these things, good things though they be, are not essential to OOP.) What makes a languages object-oriented? (Language support for this sort of packaging; you can do OOP in C or ML, but the language won't help you.)

2. Why is OOP good? (Powerful language support for thorough encapsulation; ease in modeling real-word objects / using a metaphor; ease in reusing code wisely.)

3. Define *object*, *method*, *method signature*, *method implementation*, *method invocation*, *class*. These must be good definitions. You are permitted to suggest more than one definition for these.

4. The first edition of our textbook had the following defintion:

> A client of a class C is any program that declares a variable of type C. Such a variable is called an *object*.

In the current edition, this has been changed to (pg 317):

> A *client* of a class C is any other class or method that declares or uses an object of class C. The declaration and initialization of such a variable in Java creates an *object* in the heap, since Java uses reference semantics for objects.

One can almost hear the very complaints they got about the first edition's "definition" which prompted this change. However, they still didn't get it right. Critique and correct this. (The way they changed it between editions gives a hint about what's wrong.)

5. Informally, what needs to be done to type-check an OJay program? (For each class, each declaration must have a valid type and each method must verify. For any invocation, the class of the type of the receiver must have a method of the name of the method being invoked, and the types of the formal parameters must match the actual parameters.)

6. How would you translate the following from OJay to RecJay?

```
class A {
  private int x;
  public int m(int y) { return x + y; }
}

...

    A a;
    a = new A();
    ...
    a.m(5);
```

7. How would you translate the following from POJay to OJay?

```
interface I {
   int m(int y);
}

class A implements I {
   private int x;
   public int m(int y) { return x + y;  }
}

class B implements I {
   private int xx;
   public int m(int x) { return x * y; }
}

...

   I i;
   if ( ... )
      i = new A();
   else
      i = new B();

   ...

   i.m(5);
```

8. Some languages with polymorphic types have a construct which we will call `typeswitch`, which works like a `switch` statement, except it branches based on the dynamic type of the test expression. For example, we might have in a Java- or POJay-like language

```
interface I {  . . .  }

class A implements I {  . . .  }
class B implements I {  . . .  }
class C implements I {  . . .  }
```

```
 I i;
. . .
typeswitch(i) {
   case A:
           ...
   case B:
           ...
   case C:
           ...
   default:
           ...
 }
```

Describe a way to translate such a program into standard Java (not just a Jay variant; you may assume your target language has all the features of Java available). If you like, you may assume that all cases end in a `break` (but state this assumption if you make it). If you can think of more than one way to encode this in Java (and there are several ways), you are encouraged to describe all of them. You will not be penalized for an incorrect solution if you also have provided a correct solution. Translations that work only under restricted circumstances will also be considered, but in that case you should state the restrictions.

C. Functional programming

9. What is alpha conversion? What is beta reduction? Be able to reduce (evaluate) programs/expressions in the lambda calculus. Some examples:

a. $(\lambda f\ .\ \lambda x\ .\ \lambda y\ .\ (f\ x)\ y)(\lambda w\ .\ w)(\lambda a\ .\ a)(\lambda b\ .\ b)$.

b. $(\lambda x\ .\ \lambda y\ .\ (x\ y))(\lambda z\ .\ \lambda w\ .\ w)(\lambda a\ .\ a)$

c. $(\lambda f\ .\ \lambda s\ .\ \lambda b\ .\ ((b\ x)y))(\lambda m\ .\ m)(\lambda n\ .\ n)(\lambda c\ .\ \lambda d\ .\ c)$

10. What is a free variable? What is a bound variable? (You need only define one of these; then you can define the other as being "a variable that is not [the other].")

11. In the context of a lambda-calculus program or an Em-family program, be able to use the terms *application* and *abstraction* correctly.

12. Consider the following variant of LEm with pairs.

$$\begin{array}{rcl} e & \rightarrow & x \mid \texttt{fn}\ (x)\ \texttt{=>}\ e \mid e(e) \mid \\ & & c \mid e + e \mid \\ & & (e, e) \mid \#1(e) \mid \#2(e) \end{array}$$

Pairs may contain any sort of thing, including other pairs. The standard way to write a pair type is $\tau_1 \times \tau_1$.

a. Design a type system (a grammar of types and a set of typing rules) for this language.

b. Give operational semantic rules for this language.

c. Prove Lemma 1 for the case where $e = \#1(e)$.

13. Consider the following variant of LEm with two security primitives.

$$\begin{array}{rcl} e & \rightarrow & x \mid \texttt{fn}\ (x)\ \texttt{=>}\ e \mid e(e) \mid \\ & & c \mid e + e \mid \\ & & \texttt{encrypt}\ (e) \mid \texttt{decrypt}\ (e) \end{array}$$

c ranges over (plaintext) integer constants. Both integers and functions can be encrypted, and it is possible to encrypt something already encrypted. Addition may not be performed on encrypted

integers, nor can encrypted functions be applied. The primitives `encrypt` and `decrypt` satisfy the equation

$$\texttt{decrypt}(\texttt{encrypt}(e)) = e$$

a. Design a type system (a grammar of types and a set of typing rules) for this language.

b. Give operational semantic rules for this language.

c. Prove Lemma 1 for the case where $e = \#1(e)$.

14. Using a type derivation, prove that `fn(f, x) => if f(x) then f else fn(y) => (y andalso f(y))` has type $(\texttt{bool} \to \texttt{bool}, \texttt{bool}) \to (\texttt{bool} \to \texttt{bool})$

15. Be able to transform an Em program to TailEm. Some examples:

a. `k (if x then let val y = f(x) in g(y) andalso g(x) end else h(x))`

b. `k (let val x = f(y) in fn(z) => h(x) end)`

c. `k (f(let y = h(x) in fn (z) => z(y) end))`