Computer Science 365
Final 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. *Be able to write a well-structured definition. "How a variable is stored in memory" is not a valid definition of "type."*

B. History etc. Our coverage of PL history was mainly for enrichment, but you should be able to answer these things:

1. Why are there so many programming languages? Cite two or three languages and in a sentence and state how their coexistence stems from differing purposes, differing philosophies of design (different paradigms, levels of abstraction, standardization / unification, innovation vs backward compatibility), differing origins (individuals vs committees vs corporations), or competitive differences.

2. What individual, corporation, or organization was/were behind the following languages? Plankalkul, Fortran, Cobol, Algol, LISP, APL, Pascal, Ada, C, C++, Java, C#.

C. Syntax, formal languages, and the compilation process.

3. Be able to place the following compilation phases in order and summarize in a sentence what they do: Lexical analysis, parsing (syntactic analysis), semantic analysis, optimization, code generation.

4. What is an *intermediate representation*? What representations of a program have we seen? (Original program text; token stream, parse trees / concrete syntax trees, abstract syntax trees.)

5. What is a BNF, and what are the elements of a BNF? (Rules, terminals, non-terminals, a start symbol). How does an EBNF differ from a BNF? Is EBNF notation more powerful than BNF notation?

6. Write a BNF and/or EBNF for a language like Mathex. Make sure it is unambiguous. Be able to write both left-recursive and right-recursive versions.

7. Regular expressions vs (E)BNF: How is BNF more powerful than regular expressions? What are regular expressions and BNF, respectively, typically used for in language design? Understand the significance of the fact that for most programming languages, their lexical structure is described by regular expressions (and lexical analysis algorithms are equivalent to deterministic finite acceptors), while their syntactic structure is described by a grammar (and the parsing algorithm is equivalent to a push-down automaton). Why do C and Java disallow nested block comments (i.e, `/* .... /* .... */ ... */`)?

8. Given a grammar, how would we represent it using Java classes? How using ML datatypes?

9. What does LL(1) and LR(1) mean? That is, what do the L's, R's and 1's stand for? What are LL(1) and LR(1), as nouns? (They are the names of *algorithms*.) What do we mean when we say that a *parser* "is LL(1)"? (We mean it uses the LL(1) algorithm.) What do we mean when we say that a *grammar* "is LL(1)"? What do we mean when we say that a *language* "is LL(1)"?

(I won't ask you to construct an LL(1) parsing table or do an SLR parse... that would take too much time.)

10. What do we use the Visitor pattern for? (Adding a single new operation to a set of classes; this allows us to write one new class rather than modify a large number of files). What aspect of Java are we trying to get around by using the Visitor pattern? Why don't we need the Visitor pattern in ML? How is the Visitor pattern implemented in Python? If you're using the Visitor pattern in a language project, what change are you gambling on not happening?

11. What's the difference between abstract syntax and concrete syntax? What are they, respectively, used for? Given a small grammar for concrete syntax, write an equivalent grammar for abstract syntax. Write a set of ML datatypes to represent the abstract syntax.

D. Semantics

12. Which of the following are syntactic concerns, and which are semantic concerns? Whether identifiers are correctly formed; whether a value of the right type is stored in a variable; whether a variable is declared before it is used; whether commas correctly separate a list of variables in a declaration, whether `x + y` is an integer addition or floating point addition.

13. What is a *type system*? We use it to associate what parts of the language with an appropriate type? (Answer: Variables and *expressions*. Values already are associated with types by default.) What is the difference between a *statically-typed* language and a *dynamically-typed* language? What is a *type error*? What does it mean if a program is *type safe*? What does it mean if a language is *strongly-typed*?

14. What does it mean for a Jay program to be type safe? That is, give an intuitive list of rules and conditions that must be satisfied, based on what things are a part of the Jay language or a language in the Jay family. We did this in class, and you won't need to regurgitate the exact list we came up with verbatim. You should, however, be able to come with most of the ideas that were in the list. (See below; I may also ask you to add to the list rules for JayJay, FunJay, 'RayJay, and RecJay.)

15. In our formal treatment of type rules, what do tm, V, and T stand for? (tm is the type map, a function from identifiers (variable names) to types; V is the verification function/predicate, which takes various language constructs and a type map and returns true or false, whether the piece of the program is correctly typed; T is the typing function, which takes an *expression* and a type map and returns the type of the expression. (Why does T take only expressions, whereas V takes any language construct (usually statements)?))

16. Determine the following formal type verification rules:

- $V(Assign(v, e),\ tm) =$

- $T(\mathit{Variable}(v),\ tm) =$
- $T(\mathit{IntLiteral}(i),\ tm) =$
- $V(\mathit{Loop}(g, b),\ tm) =$ (where g and b are the guard and body, respectively)

You will not be asked questions about operational or axiomatic semantics on the final, only denotational and (for the functional stuff) operational.

17. What do μ, γ, and σ stand for, both descriptively and formally? (For example, μ: descriptively it stands for the computer's *memory*; formally it is a function (or mapping) from indices (natural numbers, standing for memory locations) to values (stored in the memory location).) How does overriding union work? For example, what is the result of $\{\langle x, 5\rangle, \langle y, 2\rangle\} \overline{\cup} \{\langle y, 2\rangle, \langle z, 8\rangle\}$?

18. What is the difference between

```
(double) 5     and     (int) 3.7
```

What is the difference between

```
                       interface I {}
                       class C implements I ()
                       I i = new C();
(double) 5     and     (C) i;
```

19. If we added floating point types to Jay (call it FloaJay) and wanted to distinguish floating point + from integer + in the abstract syntax (that is, represent that as though they were separate operators), what would we need to do (or, what information would we need to acquire) before making that translation into abstract syntax?

20. Determine the following denotational semantics rules:

- $M(\mathit{Assign}(v, e),\ \sigma) =$
- $M(\mathit{Cond}(t, s_1, s_2),\ \sigma) =$ (where t is the test expression and s_1 and s_2 are the then- and else-clause, respectively)
- $M(\mathit{Loop}(g, b),\ \sigma) =$
- $M(\mathit{CountLoop}(\mathit{init}, g, \mathit{incr}, b),\ \sigma) =$

21. Suppose Jav is a language exactly like Java except it has no for loop. What would the following for loop compile to if we were doing a source-to-source compilation from Java to Jav?

```
for (int i = 0; i < 5; i++)
  System.out.println(i);
```

22. Suppose you were going to write an interpreter for FunJay by writing a translator from FunJay to JayJay and just use the JayJay interpreter. How would you compile a method call? What feature of function semantics would your solution not support?

23. Describe in a sentence or two the semantics of a function call in FunJay in terms of its effects on μ, γ, and a (the top of the stack). for call-by-value, call-by-reference, call-by-value-result, and call-by-name.

24. What are the limitations of using a standard method invocation stack? What feature does it not support?

25. What is the difference between a reference and a pointer? (Saying "you can do arithmetic on pointers" will get you most, but not full, credit.)

26. Extend Jay to a new language that includes pointers (specifically, variables may be pointer types, and the operations `*` and `&` are defined). Work out a type system (appropriate T and V functions) and denotational semantics (appropriate M functions, showing the effects on μ and γ as appropriate). *(This is actrually the PoiJay project—we already did this.)*

27. Describe informally what is necessary to evaluate qualified variables and left hand sides in RecJay. What information does the type checker need to discover and store, and how is that information used at runtime?

28. Suppose we were to extend RecJay to have exception handling (NullPointerException, ArithmeticExcaption, and ArrayIndexOutOfBoundsException; plus try/catch blocks). Describe the algorithm of a static analysis which would check to see if an exception is possible within a try block. For example, the following would be rejected because the exception is impossible:

```
...

try {
     z = p();
     try {
        x = y / z;
     } catch(ArithmeticException ae) {
         System.out.println(z);
     } catch(ArrayIndexOutOfBoundsException aioobe) {
          System.out.println(y);
     }
} catch(ArithmeticException ae) {
    System.out.println(x);

}
```

The AIOOBE is impossible because there is no array reference in the try block. The second AE is impossible because, although there is a division operation in the try block, it is already handled by a nested try/catch.

E. Object-oriented programming.

29. 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.)

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

31. 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.

32. A textbook I used to use in this course 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 a later edition, this has been changed to:

> 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.)

33. 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.)

34. 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);
```

35. 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 y) { return xx * y; }
}

...

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

   ...

   i.m(5);
```

36. 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.

F. Functional programming

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

38. 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].")

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

40. 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 $e = \#1(e)$.

41. 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 $e = \#1(e)$.

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

43. 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))`