CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 27— Course Wrap-Up

Goals for today

- (Discuss mark-sweep garbage collection from last time)
- Describe some things we didn't get to
  - Not on the final
- Review key concepts/principles we did do and put them in context

If we had 1 more OO lecture

```
class C {
   void m(A a, B b);
   void m(E e, F f);
   void m(F f, E e);
}
```

How to resolve a call `e0.m(e1,e2)`.

- *Static overloading:* Use the (compile-time) type of `e1` and `e2`.
- *Multimethods:* Use the (run-time) class of what `e1` and `e2` evaluate to.

Java/C++ have static overloading.

Both semantics can have "no best match" errors since there may be multiple methods that "match" but using different subsumptions.

What else?

Are all programming languages imperative, OO, or FP? No.

- Logic languages (e.g., Prolog)
- Scripting languages (Perl, Python, Ruby (as typically used))
- Query languages (SQL)
- Purely functional languages (no `ref` or `set!`)
- Visual languages, spreadsheet languages, GUI-builders, text-formatters, hardware-synthesis, ...
- And most languages now have support for parallel programming

Prolog in one example

```
append(nil, Lst2, Lst2).
append(cons(Hd,Tl), Lst2, cons(Hd,Tl2)) :=
     append(Tl, Lst2, Tl2).

append(cons(1, cons(2, nil)), cons(3, cons(4, nil)), X)
% X = cons(1,cons(2,cons(3,cons(4,nil))))
append(cons(1, nil), cons(2,nil), cons(1, cons(2, nil)))
% yes
append(nil, cons(2,nil), cons(1, cons(2, nil)))
% no
append(cons(Hd,nil), Y, cons(1, cons(2, cons(3, nil))) )
% Hd = 1  Y = cons(2,cons(3,nil))
```

Prolog key ideas

- A program is a set of declarative proof rules.
- Operationally, it's like a function that doesn't distinguish inputs from outputs.
- The implementation searches for the minimal constraints necessary for a formula to be true.
- Different "queries" can run "forward" or "backward"
- This is Turing-complete; killer app is inherently search-oriented tasks, which are common in AI.

Scripting Languages

Few "new" language constructs, but convenience for some quick-and-dirty programs.

- File-system access very lightweight
- Lots of support for string-processing via regular expressions (a different "pattern-matching")
- Tend to have very few "errors" (array resizing, implicit variable declaration, etc.)

Opinion:

- A fine tool for *small* tasks
- They tend to hide bugs rather than prevent them
- But you should learn to automate repetitive tasks!

Query Languages

Canonical example: Suppose there's a big database and many people need data from it. We could make lots of copies or let people submit queries.

Key idea: Move the code to the data, not the data to the code.

Interestingly: We do not necessarily want the query language to be as powerful as a Turing-machine!

SQL was carefully designed so every query terminates.

Purely Functional Languages

Example: Haskell

To make life without refs palatable, the default is "lazy" (call-by-need) evaluation.

One-line example: `let ones = 1::ones`

Laziness can lead to elegant programming and really increases the number of equivalent programs. In Haskell, `(f x) + (f x)` and `(f x) * 2` are contextually equivalent, always.

- Haskell does have *monads*, which allow a more imperative style.
- The implementation of laziness uses mutation, but in a controlled way (we did this in Scheme).

Parallelism

(As now discussed in 303/451, but it's a PL topic also), sometimes you want multiple call stacks:

- For performance (especially with multicore)
- For structuring an application

The key questions are how to thread *communicate* and how do they *synchronize*.

Easily a course in itself to learn different *parallel programming models*.

But we still did a lot

A thorough understanding of higher-order programming, variable scope, semantics of FP and OO, important idioms, static typing, ...

Oh, and you learned a healthy amount of 3 new languages.

Hopefully:

- The time you need to "pick up" a language will drop dramatically (though you have to learn big libraries too)
- You will use mutation for what it's good for and not to create brittle programs with lots of unseen dependencies
- Understand syntax matters, but it's not that interesting
- Apply idioms in languages other than where you learned them
- Recognize language-design is hard and semantics should not be treated lightly.

Top 12 Concepts?

1. Code evaluates in environments – scope/resolution matters
2. Recursive data is processed with recursive functions
3. Without mutation, copying vs. aliasing is indistinguishable
4. Closures have many powerful uses.
5. Each-of vs. one-of
6. (Dis)Advantages of static typing – (and what is checked)
7. *When* evaluation occurs is important (see thunking/macros)
8. OO vs. FP: many similarities and a couple big differences
9. Parametric polymorphism vs. subtyping
10. Function-argument subtyping is contravariant
11. Can *embed* a language in another via constructors and interpreters
12. Languages themselves are rich recursive definitions

Context

In most courses and jobs, a programming language is just a means to an end (and only one of many means).

This course was perhaps your one chance to study languages as designs that are *themselves* fascinating, beautiful, and sometimes awkward

- And there's much more to learn (441?)

I believe this makes you a better programmer, even if the rest of your life is spent in Java and C (which it won't be)