CS2135: Recap on Languages

Kathi Fisler, WPI

February 16, 2004

Now that we've spent several lectures on languages and even defined a couple, let's take stock and remind ourselves how we got here and why.

1 In the Beginning ...

On the first day of the course, as well as in your early programming experience, we wrote simple expressions that were identical minus a simple piece of information (like a number or symbol). We created functions with parameters to let us reuse the common code:

```
(* 12 (/ 3 8))
(* 15.60 (/ 3 8))

(define (share-cost price)
  (* price (/ 3 8)))
```

Functions are perhaps the most fundamental form of reuse, as well as the simplest to understand.

2 Map and Filter

Map and filter took us to a new level of reuse. Previously, we used functions to allow reuse over data like numbers, strings, and lists. Map and filter came about because we realized that code is often identical except for some *function* (instead of some data). With map and filter, we explored taking functions as parameters. It's the same basic idea that you initially learned for parameters and functions, but what you can pass as a parameter got more interesting. As a result, the reuse is more powerful.

```
;; boa-foods : list[boa] → list[symbol]
;; return list of foods eaten by boas in input list
(define (all-boa-foods alob)
  (cond [(empty? alob) empty]
        [(cons? alob) (cons (boa-food (first alob))
                            (all-boa-foods (rest alob)))]))

;; tiger-lengths : list[tiger] → list[number]
;; return list of lengths of all tigers
(define (all-tiger-lengths alot)
  (cond [(empty? alot) empty]
        [(cons? alot) (cons (tiger-length (first alot))
                            (all-tiger-lengths (rest alot)))]))

;; map : (alpha -¿ beta) list[alpha] → list[beta]
;; return list of results of running function on each item in input list
(define (map f alst)
  (cond [(empty? alst) empty]
```

```
        [(cons? alst) (cons (f (first alst))
                            (map f (rest alst)))]))

(define (all-boa-foods alob)
  (map boa-food alob))

(define (all-tiger-lengths alot)
  (map tiger-length alot))
```

3 Languages

The naïve versions of the powerpoint and tax programs showed yet another example of condensing and reusing common code. Here, we collapsed common sequences of instructions into commands and used them to build a language. As a reminder, we started with the following naïve powerpoint program:

```
(begin
  (print-string "————————————————————")
  (print-string "Hand Evals in DrScheme")
  (print-string "Hand evaluation helps you learn how Scheme reduces programs to values")
  (print-string "————————————————————")
  (await-click)
  (print-string "————————————————————")
  (print-string "Example 1")
  (print-string "(+ (* 2 3) 6)")
  (print-string "(+ 6 6)")
  (print-string "12")
  (print-string "————————————————————")
  (await-click)
  (print-string "————————————————————")
  (print-string "Summary: How to Hand Eval")
  (print-string "Find the innermost expression")
  (print-string "Evaluate one step")
  (print-string "Repeat until have a value")
  (print-string "————————————————————")
  )
```

We marked off the common patterns, turned them into co,,amds, and wrote an interpreter to run those commands and reproduce our slideshow program. At that point, we had defined a language.

But what was special about this case? Why did powerpoint get called a language, while map and filter were just helper functions? The process of creating language commands looks very similar to the process of creating helper functions. We could have just made a *print-slide* helper function for the common code to get rid of the repetition. Why did we get into all of this languages stuff?

A couple of differences appear in the code that results from creating helper functions versus defining languages.

- When we create helper functions, we may change *some* of our code (the part that used the common pattern), but the rest of it stays the same. When we create a language, eventually all of the code migrates to the new language.
- As we add more to a program that uses helper functions, we may call any functions defined in that file. As we add code to a program that uses a language, we only use the constructs of that language.

These observations suggest that the difference between when you've made a language versus just made helper functions has to do with whether you restrict subsequent code to use *only* the helper functions (aka the language). Why would anyone want to do this though?[1]

[1] One benefit is that once we have a language, we can reimplement the language/interpreter in any language we choose. So we can start with a Scheme version of a powerpoint program and migrate that into a powerpoint language implemented in C.

The Real Point of a Language

Languages allow us to restrict what a programmer can do.

A programmer working in our powerpoint language is given limited powers: to create slides with certain attributes and to display them. We control how the slides display and when the interpreter should wait for clicks. Imagine that instead of creating a language, we had just added a bunch of helper functions to the powerpoint file and told the programmer to work with those. The programmer could now create powerpoint programs that wait for multiple clicks per slide, or display slides upside down, or gosh knows what else. By giving the programmer a restricted set of commands, we control what programs the programmer can write at all.

Why is this a good thing? Don't we want programmers to have as much flexibility as possible? Why shouldn't a programmer be allowed to expect two clicks if she wants to? Programming is all about freedom of expression and control of the machine (that's one reason why it's so much fun). Restrictions in languages are clearly anti-programmers, no?

All languages embody some restrictions. Should the powerpoint programmer be able to crash the machine? Edit your password file? Take down the network? Of course not. The real question (and perhaps the hardest thing in real language design) is choosing the *right* set of restrictions. If you recall our comparison of languages in the first languages lecture, we showed how Fortran allowed goto statements but later languages restrict how to move around programs; C++ allows the programmer access to pointers while Scheme does not. Every language is a combination of what they help a programmer to do, and what they disallow in the process.

Since this is an intro class, we're not getting into the topic of how to decide which restrictions to put in a language (this is a fascinating topic though). We're simply learning the mechanics of creating languages and writing interpreters to run languages. You can build on this background to restrict languages as you see fit.

4 Summary

At a core level, much of this course is about one topic: creating ways to reuse common code. Now that we're in the languages portions, we're just doing more complex and interesting forms of reuse, but it is all about reuse. If you're feeling lost in this section of the course, see if you can identify what is getting reused in the exercises that we do. If you don't see that, come ask one of us for help.