University of Virginia 19 October 2011
cs1120: Introduction of Computing
Explorations in Language, Logic, and Machines

Class 24: Imperative Programming

Upcoming Schedule

- **Office hours:**
  Today: 5-6:30pm (Jiamin, Rice 1st)
  Thursday: 9:45-11am (Dave, Rice 507); 1-2:30pm/4:30-7:30pm (Rice 1st floor)
- **Friday, 21 October (beginning of class): Problem Set 5 (see below)**
- **Reading:** if you haven't already, should soon finish reading the course book through the end of Chapter 9, and *The Information* through the end of Chapter 7.
- **Wednesday, 26 October: Quiz 3** (covers the course book through Chapter 10, *The Information* through the end of Chapter 8, and some other readings, details to be provided later)

Functional Programming: divide a problem into *procedures* that can be *composed* to solve problem. Procedures are connected by passing the results of one procedure as the inputs to the next procedure. The way the functions are composed determines the order in which expressions are evaluated.

Imperative Programming: divide a problem into *statements* (mostly assignments) that can be done in order to solve a problem. Statements are connected by using shared state.

Very little programming is *purely functional* or *purely imperative*, but most is a mix of both. Almost everything we have done until PS5 was purely functional, but we still needed some state and side-effects (like printing on the screen).

Places: a place stores a value
Frame: a collection of zero or more places (like a picture frame)
Environment: a **frame**, and a pointer to a parent environment
Global Environment: the only environment with no parent

define creates a new place in the execution environment's frame, and stores a value in that place
set! ("set bang") changes the value associated with a place

Applying a procedure creates a new environment, and creates places in that environment's frame named after the parameters, with the argument values.

Mutable Pairs

(mcons *a b*) creates a new mutable pair
(mcar *p*) selects the first part of a mutable pair, **(mcdr *p*)** selects the second part of a mutable pair
(set-mcar! *p v*) assigns the value of *v* to the first part of the mutable pair *p*
(set-mcdr! *p v*) assigns the value of *v* to the second part of the mutable pair *p*

```
(define (list-append p q)
  (if (null? p) q
      (cons
       (car p)
       (list-append (cdr p) q))))
```

What is the running time of **list-append**?

What is the memory use of **list-append**?

```
(define (mlist-append! p q)
  (if (null? p)
      _____
      (if (null? (mcdr p))
          (set-mcdr! p q)
          (mlist-append! (mcdr p) q))))
```

What is the running time of **mlist-append!**?

What is the memory use of **mlist-append!**?

Define a procedure **make-list** that takes as input a number **n**, and outputs a list of length *n*.

What is the running time of **make-list**? Also, state it in terms of the *size* (not value) of the input.

Aliasing: when two objects share places.

Define a **mlist-reverse!** that reverses the elements of a mutable list. The output should be a mutable list with the elements in the reverse order. The number of cons cells it creates should not scale with the length of the list. (The input list can be mutated arbitrarily!)