More Scheme

CS152

Chris Pollett

Nov. 3, 2008.

Outline

- More Scheme

Introduction

- Last day, we were talking about pure functional programming languages, one of the main features of which is that such a language does not have variables.
- We talked about a function being referentially transparent if its values only depends on its arguments; I.e., no static variables.
- Such pure languages are easier to prove correctness properties of their programs.
- We said although no language is strictly pure some languages like Scheme and ML are closer to purely functional.
- We introduced Scheme, talked about expression, evaluation, if, and cond.

Boolean Operations

- There are a number of functions which can be used to compare objects in Scheme.
- For numbers, we have already seen <, >, <=, >=. For example, (< 2 3) returns #t
- For equality one have eq?, eqv?, and equal? The first is somewhat implementation dependent, the second checks if the two items are equivalent (essentially refer to the same value), the last does a deep comparison.
- Notice the convention that "?" is used after functions which might return a true or false.
- Other, boolean functions: null?, string=?, number?, etc.
- Scheme also has the Boolean operators and, or, not.

let and begin

- Scheme has function called let which allows values to be given temporary names within an expression:

  (let ((a 2) (b 3)) (+ a b)) ; evaluates to 5
- The first expression within let is called a **binding list**.
- You could view let as short for

  ((lambda (a b) (+ a b)) 2 3)
- You could have multiple expressions not just (+ a b) at the end of a let (you can also do this with lambda)
- If you don't need to do any assignments, another useful Scheme operation which essentially does this is begin:

  (begin l1 l2 l3 …)

  Executes l1 followed by l2 followed l3. You could view this as short for:

  ((lambda () l1 l2 l3 …))

Adding things to the Scheme Environment

- The define function can be used to add new associations between names and values in Scheme:

```
(define a 2)
(define emptylist '())
(define (sum lo hi) ; could write: sum (lambda (lo hi) …
   (if (= lo hi)
       lo
       (+ lo (sum (+ lo 1) hi))) )
```

- Once something has been defined, you can see its value are scheme prompt

➢ (sum 3 5)

12

Data Structures in Scheme

- The basic data structure in Scheme is the list.
- List can be nested to create more complicated structures.
- For instance, ((a b) (c d) e).
- There are three built-in functions to manipulate lists: car, cdr, cons -- the names correspond to assembly instructions on a now defunct IBM mainframe.
  - (car '(1 2 3)) ; returns 1 - the head of the list
  - (cdr '(1 2 3)) ; returns (2 3) the tail of the list
  - (cons 0 '(1 2 3)) ; adds to the front of the list (0 1 2 3 4)
- What is (cons 'a 'b) ? Notice 'b is not a list. It is a so-called dotted pair: (a . b).
- In general, a list is short for the coresponding nesting of dotted pair that begins with '(). So (b) is (b . '())
- The function list can be used to make a list out of a collection of objects (list 'a 'b 'c) makes the list (a b c).

Doing Things Recursively

- One can use recursion to do an operation on all the elements of a list or data structures derived from it:

```
(define (my-reverse-list L)
   (if (null? L) '()
      (append (my-reverse-list (cdr L)) (list car L))))
```

- Recursion is often viewed as wasteful of space and time because it requires storage on the stack, and the push and popping from the stack takes times.
- Modern translators can recognize certain kinds of recursions as being amenable for conversion to loops: those where the last operation in the procedure is a call to itself. (**a tail recursive procedure**).

Doing Things Recursively 2

- One trick for making a procedure tail recursive is make use of an auxiliary function with an extra accumulating parameter:

```
(define (reverse-aux L list-so-far)
  (if (null? L) list-so-far
       (reverse-aux (cdr L) (cons (car L) list-
  so-far)))) ; tail-recursive
(define (my-reverse L) (reverse-aux L '()))
```

Higher Order Functions

- We said that functional programming languages allow us to treat functions as first class objects.
- We've already seen one way that Scheme allows this: We can talk about functions independent of assigning them a name using lambda:

((lambda (x) (* x x)) 3) ; gives 9

- We can also write functions that take functions as arguments:

  (define compose (lambda (f g) (lambda (x) (g (f x)))))

- There are some built in functions of this kind. For instance, (map f L) applies the function f to each element of L.

Notions of static in Scheme

- Consider

```
(define make-new-balance (lambda (balance)
   (lambda (amount)
      (if (< balance amount) "insufficient funds"
      (begin
         (set! balance (- balance amount))
         balance)))))
(define atm (make-new-balance 100))
(atm 20)
; returns 80
```

- You can think of set! as altering the list associated with the function returned by make-new-balance changing the value stored for balance within it. I.e., balance, is acting as a static variable and this is achieved using higher-order functions.

Message Passing

- Since we can now do static variables in Scheme, we can create classes using the message passing techniques we have talked about before.
- A good example of doing this, can be found in some of my example code for Hw1 the last time I taught AI:

http://www.cs.sjsu.edu/faculty/pollett/156.1.04s/index.html?Hw1.shtml