CS 603: Programming Language Organization

Lecture 11

Spring 2004

Department of Computer Science

University of Alabama

Joel Jones

Outline

- Questions
- μ-Scheme (cont.)
- Reading for next time

Larger LISP Example

- Calculate prime numbers less than *n* using Sieve of Eratosthenes.

Let's Play at the Board
(but your books can't play)

- Insertion sort–given a list of *n* elements, sort the last *n-1* recursively, then insert the first in its proper position.
- `(define insert (x l) …`
- `(define insertion-sort (l) …`

Association Lists

- association list(a-list)–a list of the form $((k_1\ a_1), \ldots, (k_m\ a_m))$, where the k_i are symbols (called *keys*) and the a_i are *attributes*.
- retrieve data:

```
(define assoc (x alist)
  (if (null? alist) '()
    (if (= x (caar alist)) (cadar alist)
      (assoc x (cdr alist)))))
```

Association Lists (cont.)

- add data

```
(define mkassoc (x y alist)
  (if (null? alist)
    (list1 (list2 x y))
    (if (= x (caar alist)
      (cons (list2 x y) (cdr alist))
      (cons (car alist) (mkassoc x y (cdr
alist))))))
```

Let's play at the board again (No Books!)

- Property lists–a-list where attribute is an attribute list
- Examples
  - `(set fruits '((apple ((texture crunch)))`
    `(banana ((color yellow)))))`
  - `(getprop 'apple 'texture fruits)`
    `crunchy`
- Write `(getprop x p plist)`, where x is the individual, p is the property and plist is the property list

Let's play at the board again (No Books!)

- `(putprop x p y plist)` give individual x value y for property p in plist
  - `(set fruits (putprop 'apple 'color 'red fruits))`
    `->((apple ((texture crunchy)(color red)))(banana ((color yellow))))`

μ-Scheme

- Closures
  - Pair of lambda (function value) and environment
    - «`(lambda (y) …)`, {x |-> *l*}»
  - Environment maps name to mutable location

```
->(val counter-from
       (lambda (n)
           (lambda () (set n (+ n 1)))))
->(val ten (counter-from 10))
<procedure>
->(ten)
11
->(ten)
12
```

μ-Scheme (cont.)

- Closures

Pair Up:

• Write a function (make-withdraw) that models a bank account balance, where only withdrawals are allowed

```
->(val make-withdraw (lambda (balance) …))

->(val W1 (make-withdraw 100))

->(W1 10)

90
```

Simple higher-order functions

- Composition
  - (define o (f g) (lambda (x) (f (g x))))
  - (define even? (n) (= 0 (mod n 2)))
  - (val odd? (o not even?))

Pair Up:

• Write a function (to8th) using composition with square and ? that raises the input to the 8th power

```
-> (define square(x) (* x x))
square
-> (val to8th …)
<procedure>
-> (to8th 2)
256
```

Higher-order functions on lists

- Filter –

```
(define filter (p? l)
    (if (null? l) '()
        (if (p? (car l))
            (cons (car l) (filter p? (cdr l)))
            (filter p? (cdr l)))))
```

- Exists? –

```
(define exists? (p? l)
    (if (null? l) #f
        (if (p? (car l))
            #t
            (exists? p? (cdr l)))))
```

Higher-order functions on lists (cont.)

- All? –

```
(define all? (p? l)
    (if (null? l) #t
        (if (p? (car l))
            (exists? p? (cdr l))
            #f)))
```

- Map –

```
(define map (f l)
    (if (null? l) '()
        (cons (f (car l)) (map f (cdr l)))))
```

Pair Up:

- How are the arguments of `filter`, `exists?`, and `all?` different from those of `map?`

Higher-order functions on lists (cont.)

- Foldr
- Foldl

Simple higher-order functions (cont.)

- Currying
  - Taking an *n*-argument function and turning it into a 1-argument function returning a function expecting *n*-1 arguments (also curried!)
  - Currying binary functions
    - (define curry (f) (lambda (x) (lambda (y) (f x y))))
    - (define uncurry (f) (lambda (x y) ((f x) y)))

Simple higher-order functions (cont.)

- Currying

```
->(val zero? ((curry =) 0))
->(zero? 0)
#t
->(val add1 ((curry +) 1))
->(add1 4)
5
->(val new+ (uncurry (curry +)))
->(new+ 1 4)
5
```

Higher-order functions for polymorphism

- Implementing sets
  - Constructing mono-morphic sets requires definition of equality—simple if primitive elements

```
->(val emptyset '())
->(define member? (x s) (exists? ((curry equal? x) s))
->(define add-element (x s) (if (member? X s) s (cons x s)))
->(define union (s1 s2) (foldl add-element s1 s2))
->(define set-from-list (l) (foldl add-element '() l))
```

  - Problem is equality for complex types

Higher-order functions for polymorphism (cont.)

Pair Up:

• Implement a data type for sets which uses the usual list representation. Therefore, all you have to do is implement `=set?`

```
->(=set? '(a b c) '(c b a))
#t
```

• Implement sets of sets, where the set functions are passed an eqfun for comparing sets.

```
->(define member? (x s eqfun) …)
->(define add-element (x s eqfun) …)
```

Higher-order functions for polymorphism (cont.)

- How to construct?
  - Add parameter to *every* function
  - Make part of "object" (set in previous example)
  - Make list of functions (like C++ templates)

Reading & Questions for Next Class

- Chapter 3.11–3.14