6.001 Tutorial 4 Notes

Gerald Dalley
28–29 Feb 2005

Announcements

- Quiz Wednesday, 2 Mar. 2005, 7:30-9:30pm.
  - One page of notes.
  - Last name starts with A-M → 32-123
  - Last name starts with N-Z → 34-101
  - Review session Monday 28 Feb, 8:30–10:30pm in 32-D463.
  - See course website for details, old quizzes.
  - Office hours (me) 28 Feb, 4:00-5:30pm in 32-044F. Bring your own questions.
- No recitation on Wednesday (happy cramming!)

Higher-Order Procedures

Higher-order procedures are procedures that either accept procedures as arguments or return procedures as their values.

```
;; This procedure curries a function,
;; e.g. it takes a function of two
;; inputs and returns a function of
;; one input, that returns a function
;; of one input, that does the same
;; thing
;; ex: (+ 1 2) ==> 3
;; (((curry +) 1) 2) ==> 3
(define (curry f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
;; This procedure composes two
;; functions f and g, each of one
;; argument, and returns a procedure
;; of one arg that does (f (g x))
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

```
;; This procedure applies f to each
;; element of the list, and returns a
;; new list made from those values
(define (map f lst)
  (if (null? lst)
      nil
      (cons (f (car lst))
            (map f (cdr lst)))))
```

```
;; This procedure returns a new list
;; containing the elements in the
;; original for which pred is true
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
;; This procedure combines all the
;; elements of lst using the binary
;; operation op, terminating with init
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))
```

Practice with HOPs

Suppose `lst` is bound to the list (1 2 3 4 5 6 7). Using `map`, `filter`, and/or `fold-right`, write an expression involving `lst` that returns:

`(1 4 9 16 25 36 49)`

```
(map square lst)
```

`(1 3 5 7)`

```
(filter odd?  lst)
```

`((1 1) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7))`

```
(map (lambda (x) (list x x)) lst)
```

The maximum element of `lst`: 7

```
(fold-right max (car lst) (cdr lst))
```

`((2) ((4) ((6) #f)))`

```
  (fold-right (lambda (x y)
                 (list (list x) y))
              nil
              (filter even?  lst))
```
Hint: this list may be constructed via
```
(list (list 2)
      (list (list 4)
            (list (list 6)
                  nil)))
```

The last pair of `lst`: (7)

Impossible! `map`, `filter`, and `fold-right` only give you access to the *members* of the list, not the *backbone* – the cons cells which make up the list.

Data Abstraction: Sets

A set is an unordered collection of items, where each item may occur at most 1 time in the set. Adding the same item multiple times to a set does not change the set.

How should we represent a set?

1. As an unordered list (what we'll do)
2. As a sorted list (can be convenient)
3. As a sorted tree (efficient)

For the following problems, assume that the basic list ops, `filter`, `fold-right`, `map`, `filter`, `append`, `length`, and `sort` are available.

```
;; Special value: empty set
;; Represents a set with no elements
(define empty-set
  nil)
```

```
;; Evaluates to #f if the element elm is not
;; contained in the set.  Depends on the
;; internal representation of set.
(define (set-contains? elm set)
  (cond ((null? set) #f)
        ((= (car set) elm) #t)
        (else (set-contains? elm (cdr set)))))
```

```
;; Adds elm to the set if it is not already
;; part of the set.  Evaluates to the
;; new set.
;; Relies on the internal representation.
(define (set-add elm set)
  (if (set-contains? elm set)
      set
      (cons elm set)))
```

```
;; Converts a list, lst, into a set.
;; Representation-independent.
(define (list-to-set lst)
  (fold-right set-add empty-set lst))
```

```
;; Converts a set into a list
;; Representation-independent.
(define (set-to-list set)
  set)
```

```
;; Takes the original set and removes elm
;; and returns the new set.  Evaluates to
;; the original set if elm was not present.
;; Representation independent.
(define (set-remove elm set)
  (list-to-set
   (filter (lambda (x) (not (= elm x)))
           (set-to-list set))))
```

```
;; Evaluates to a set that contains all
;; elements present in either s1, s2, or
;; both.
;; Representation independent.
(define (set-union s1 s2)
  (fold-right set-add s2 (set-to-list s1)))
```

```
;; Evaluates to the set containing only
;; elements common to both s1 and s2.
;; Representation independent.
(define (set-intersection s1 s2)
  (list-to-set
    (filter (lambda (elm)
              (set-contains? elm s2))
            (set-to-list s1))))
```

```
;; Determines whether sets s1 and s2 contain
;; exactly the same sets of values.
;; Representation independent.
(define (set-eq? s1 s2)
  (define (helper l1 l2)
    (cond ((null? l1) #t)
          ((= (car l1) (car l2))
           (helper (cdr l1) (cdr l2)))
          (else #f)))
  (let ((l1 (sort (set-to-list s1) <))
        (l2 (sort (set-to-list s2) <)))
    (if (= (length l1) (length l2))
        (helper l1 l2)
        #f)))

;; Alternative implementation using an
;; advanced form of map.
(define (set-eq? s1 s2)
  (define (bool-and a b) (and a b))
  (let ((l1 (sort (set-to-list s1) <))
        (l2 (sort (set-to-list s2) <)))
    (if (= (length l1) (length l2))
        (fold-right bool-and #t (map = l1 l2))
        #f)))
```

```scheme
;; Evaluates to a set containing the
;; elements of s1 that are not present
;; in s2.
;; Representation independent.
(define (set-diff s1 s2)
  (list-to-set
   (filter (lambda (elm)
             (not (set-contains? elm s2)))
           (set-to-list s1))))
```