Programming Abstractions

Lecture 8: Variadic functions

Stephen Checkoway

Recap of `map` and `apply`

If you have a list of data and you want to apply a procedure to each element of the list, use map

```
(map f '(1 2 3)) => (list (f 1) (f 2) (f 3))
```

If you have a procedure and a list of data and you want to call the procedure with the data in the list as the arguments, use apply

```
(apply f '(1 2 3)) => (f 1 2 3)
```

If `lst` is a list of integers and you want to get a list with all of the integers doubled (i.e., `'(1 2 3)` -> `'(2 4 6)`), which should you use?

A. `(* 2 lst)`

B. `(apply (λ (x) (* 2 x)) lst)`

C. `(map (λ (x) (* 2 x)) lst)`

D. `(apply * 2 lst)`

E. `(map * 2 lst)`

If `foo` is a procedure that takes a variable number of arguments and `lst` is a list of arguments you want to pass to `foo`, how do you do it?
E.g., if `lst` is `'(a b c)`, you want to call `(foo 'a 'b 'c)`.

A. `(map foo lst)`

B. `(apply foo lst)`

C. `(map (λ (x) (apply foo x)) lst)`

D. `(apply (λ (x) (map foo x)) lst)`

E. This is not possible

Distance of a 2-d point from the origin

Recall that a point (x, y) lies $\sqrt{x^2 + y^2}$ from the origin

Let's make a procedure to compute this

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))

(distance-from-origin 3 4) => 5
```

Distance of a 2-d point from the origin

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))
```

If we have a point

```
(define p '(5 -8))
```

how can we get its distance from the origin? We can't use

```
(distance-from-origin p)
```

We can use apply

```
(apply distance-from-origin p)
```

Of course, we could also do

```
(distance-from-origin (first p) (second p))
```

Using map and apply together

Let's sum up all numbers in a structured (i.e., non-flat) list

```
(define (sum-all x)
  (cond [(number? x) x]
        [(list? x) (apply + (map sum-all x))]
        [else
         (error 'sum-all
                "~v isn't a number or list"
                x)]))
```

```
(sum-all '(1 2 (3 4 (5) () 6) 8)) => 29
(sum-all '(1 2 (x))) => sum-all: 'x isn't a number or list
```

Exercise

Write a procedure `(num-matching proc x)` that takes in a procedure proc and structured (i.e., non-flat) list and returns a count of how many elements in the list satisfy proc. For example,

```
(num-matching positive? '(1 (-2 3) (0 (4 5 -1 -2 -3)) (-1)))
```

returns 4

Hint:

```
(define (num-matching proc x)
  (cond [(list? x) ???]
        [(proc x) ???]
        [else ???]))
```

Use both `map` and `apply`

How would we implement map?

Non-tail-recursive

- Simple, clear

Tail-recursive

- Use an accumulator to hold the reversed results, then reverse

General map

`(map proc lst1 lst2 … lstn)`

If `proc` is a procedure of *n* arguments, then `map` will apply proc to corresponding elements *n* lists (which all have the same length)

```
(map f '(a b c) '(1 2 3)) => (list (f 'a 1) (f 'b 2) (f 'c 3))

(map cons '(a b c) '(x y z)) => '((a . x) (b . y) (c . z))

(map list '(a b) '(c d) '(e f)) => '((a c e) (b d f))

(map * '(0 1 2) '(3 4 5) '(6 7 8)) => '(0 28 80)
```

How would we implement the general map?

Two issues

- How do we write a procedure that takes a variable number of arguments?
- How do we apply a procedure to a variable number of arguments?
  - This one we know! Use `apply`

Variable argument procedure

`(define foo (λ params body))`

When `params` is a **list of identifiers**, the identifiers are bound to the values of the procedure's arguments

When `params` is an **identifier** (i.e., not a list), then the identifier is bound to a list of the procedure's arguments

```
(define count-args
  (λ params
    (length params)))
```

```
(count-args 'a 2 #f) => 3
```

```
(define list
  (λ elements elements))
```

Required parameters + variable parameters

`(define foo (λ (x y z . params)) body)`

Separate the required parameters from the list of variable parameters with a period

```
(define drop-2
  (λ (x y . lst) lst))
```

```
(drop-2 1 2 3 4)
```

- `x` is bound to `1`
- `y` is bound to `2`
- `lst` is bound to `'(3 4)`

Variable argument procedure with define

`(define (foo . params) body)`

```
(define (count-args . args)
  (length args))
```

With some required parameters

```
(define (drop-2 x y . others)
  others)
```

How would you write a variable-argument procedure that maps its first argument `f` over each of its other arguments and returns the result as a list? E.g.,

`(map-over add1 1 3 5 7) -> '(2 4 6 8)`

A.
```
(define (map-over f lst)
  (map f lst))
```

B.
```
(define (map-over f lst)
  (apply f lst))
```

C.
```
(define (map-over f . lst)
  (map f lst))
```

D.
```
(define (map-over f . lst)
  (apply f lst))
```

Thinking through the general map

`(map proc lst1 lst2 … lstn)`

We can use a variable-argument procedure definition for map

```
(define (map proc . lsts) …)
```

Now `lsts` is the list `(list lst1 lst2 … lstn)`

At each step of map, we need to compute

```
(proc (first lst1) (first lst2) … (first lstn))
```

The problem is we don't have a fixed number of lists, we just have a list of lists

Solution: write a procedure `map1` that just works with a single list

```
(apply proc (map1 first lsts))
```

gives a list containing the first element of each list

General map implementation

Give this a try on your own!

Hints

- Define a helper function `(map1 f lst)` that applies a single-argument procedure `f` to the elements of `lst`
- Write `(define (map proc . lsts) …)`
  - Use `map1` to get the heads and tails of elements in `lsts`
  - Use `apply` to apply `proc` to the heads and `cons` the result onto an appropriate recursive call of `map`

```
(define (map1 f lst) …)
(define (map proc . lsts)
  … (apply proc heads) …)
```

Now try making `map1` and `map` tail-recursive!