Programming Abstractions

Lecture 7: Map and Apply

Stephen Checkoway

Motivation

You have a list of data `lst` and you have a procedure `f` and you want to apply `f` to every element of `lst`, getting a new list back

- E.g., you have `'(1 2 3)` and you want `(list (f 1) (f 2) (f 3))`

Example: Adding a base to a list of offsets

Imagine we have some base value and a list of offsets and we want the result of adding the `BASE` to each of the offsets

```
(define BASE 100)
(define OFFSETS '(1 3 5 6 8 52))
```

we can write a procedure to take a list of offsets and produce a list of final values of `BASE` + `offset`: `'(101 103 105 106 108 152)`

Example: Adding a base to a list of offsets

```
(define (final-values lst)
  (cond [(empty? lst) empty]
        [else
         (let ([val (+ BASE (first lst))]
               [remainder (final-values (rest lst))])
           (cons val remainder))]))
```

Example: Getting domains from a URL

Imagine we had a list of URLs like

```
(define urls
  '("https://www.cs.oberlin.edu/classes/major-in-cs/"
    "https://checkoway.net/teaching/cs275/2021-fall/"
    "https://duckduckgo.com"))
```

and we wanted a list of domains that corresponded to those URLs

```
'("www.cs.oberlin.edu" "checkoway.net" "duckduckgo.com")
```

we could write a procedure turn a list of URLs into a list of domains

Example: Getting domains from a URL

```
(require net/url) ; defines string->url and url-host

(define (domains lst)
  (cond [(empty? lst) lst]
        [else
         (let* ([url (string->url (first lst))]
                [domain (url-host url)]
                [other-domains (domains (rest lst))])
           (cons domain other-domains))]))
```

Example: List of courses

We have a list of courses (represented as a list) like

```
(define COURSES
  '((CSCI 150 "Professor Feldman")
    (CSCI 151 "Professor Geitz")
    (CSCI 241 "Professor Hoyle")
    (MATH 220 "Professor Calcut")))
```

and we want just a list of course numbers '(150 151 241 220)

We can write a procedure to turn a list of courses into a list of numbers

Example: List of courses

```
(define (course-numbers lst)
  (cond [(empty? lst) empty]
        [else (let* ([course (first lst)]
                     [num (second course)]
                     [others (course-numbers (rest lst))])
                (cons num others))]))
```

Similarities

In each case, we have a list of elements of type α

We have an operation we want to apply that takes a value of type α and returns a value of type β

We want to apply that operation to each element of our list to get a list of elements of type β

Examples:

- Base + offset: $\alpha = \beta =$ number
- Domains: $\alpha =$ URL, $\beta =$ domain (both were strings here)
- Courses: $\alpha =$ course (as a list), $\beta =$ number

Similarities

In each case, we have

- list of α
- An operation $\alpha \to \beta$

And our output is a list of β

Map: the simple case

`(map proc lst)`

`map` applies the procedure `proc` to every element in list `lst`

```
(map f '(1 2 3 4)) => (list (f 1) (f 2) (f 3) (f 4))

(map sub1 '(10 15 20)) => '(9 14 19)

(map (λ (x) (list x x)) '(a b c)) => '((a a) (b b) (c c))

(map first '((a 5) (b 6) (c 7))) => '(a b c)
```

In each case

- `proc` is a function $\alpha \to \beta$
- `lst` is a list of α
- the result is a list of type β

Rewriting our examples with map

```
(define (final-values lst)
  (map (λ (offset) (+ BASE offset)) lst))

(define (domains lst)
  (map (λ (url)
         (url-host (string->url url)))
       lst))

(define (course-numbers lst)
  (map second lst))
```

What is the result of this?

```
(map rest '((a 5) (b 6) (c 7)))
```

A. `'((5) (6) (7))`

B. `'(5 6 7)`

C. `'((b 6) (c 7))`

D. `'(5) '(6) '(7)`

E. `'(b c)`

What is the result of this?

```
(map (λ (lst) (cons (first lst) lst))
     '((1 2) (3 4)))
```

A. `'(1 3)`

B. `'((1 1 2) (3 3 4))`

C. `'((1 (1 2)) (3 (3 4)))`

D. `'((1 4) (2 3))`

E. `'((1 3) (2 4))`

There's a standard library procedure `(round x)` that takes a number as input and rounds it to the nearest integer

If we have a list of numbers `'(1.1 2.9 3.5 4.0)` and we want a list of rounded numbers `'(1.0 3.0 4.0 4.0)`, how can we get that?

A. `(map (round x) '(1.0 3.0 4.0 4.0))`

B. `(map (λ (x) (round x)) '(1.0 3.0 4.0 4.0))`

C. `(map round '(1.0 3.0 4.0 4.0))`

D. `(round '(1.0 3.0 4.0 4.0))`

E. More than one of the above

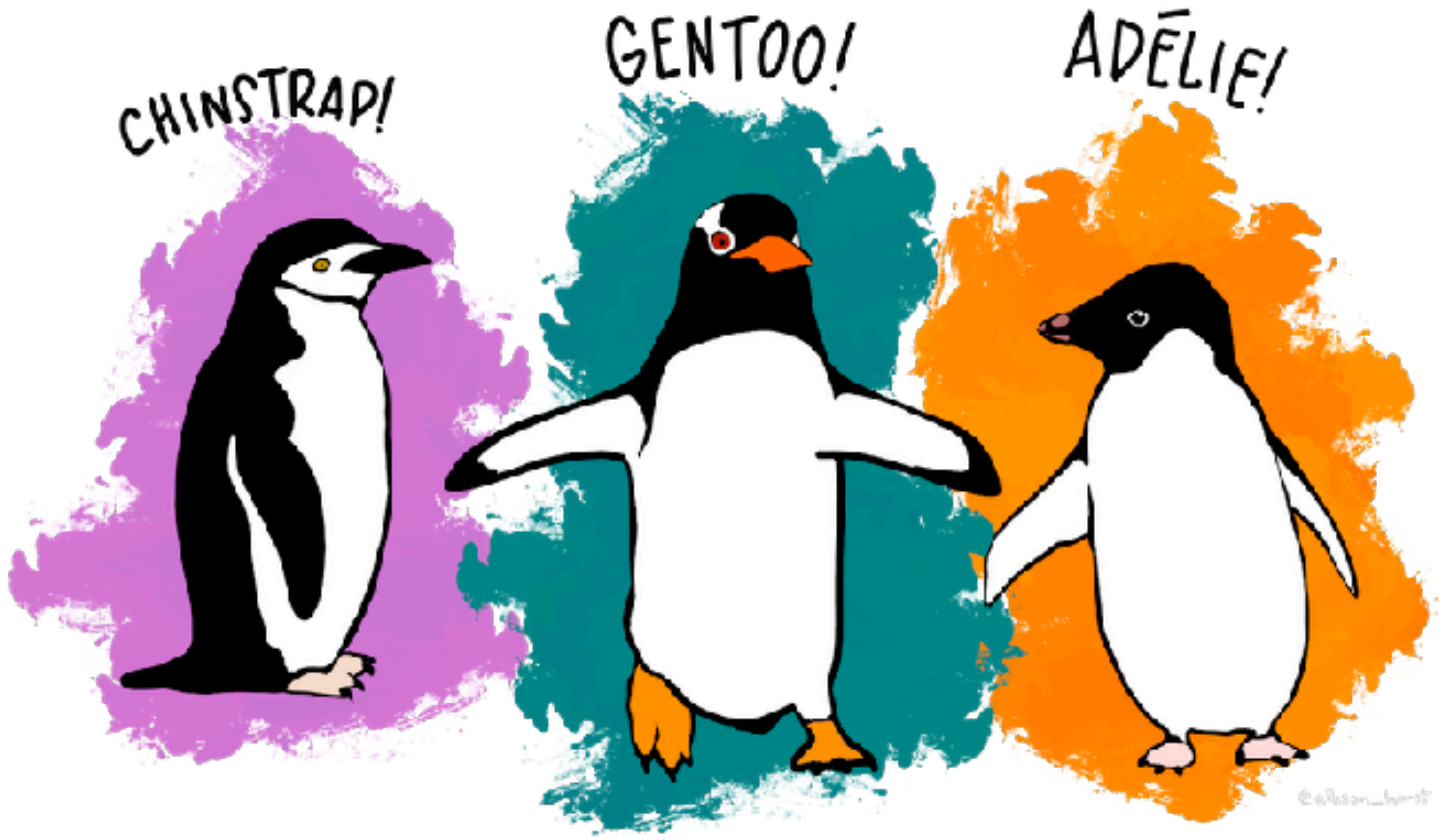
Using map to extract structured information

Imagine you had some data for penguins structured as a list of records and each record is a list:

```
(species island mass sex year)
```

E.g.,

```
(define penguins
  '((Adelie Torgersen 2750 male 2007)
    (Gentoo Biscoe 4400 female 2008)
    …))
```



We can get a list of masses of the penguins via map

```
(map third penguins) => '(2750 4400 …)
```

Get the average mass of Gentoo penguins

```
(species island mass sex year)
```

We can get a list of Gentoo penguins via filter

We can get the masses via map

```
(define average-gentoo-mass
  (let* ([gentoos
          (filter (λ (p) (eq? (first p) 'Gentoo)) penguins)]
         [masses (map third gentoos)])
    (/ (sum masses) (length gentoos))))
```

Do we have to write sum again?

We know that + takes any number of arguments, e.g., `(+ 1 5 3 -8 20)`

We have a list of masses

It'd be nice to tell Racket, "use this list as the arguments to +"

Applying a procedure to a list of arguments

`(apply proc lst)`

Applies `proc` to the arguments in `lst` and returns a single value

```
(define (maximum lst)
  (apply max lst))
(maximum '(1 3 4 2)) => (apply max '(1 3 4 2))
                     => (max 1 3 4 2)
                     => 4

(define (sum lst)
  (apply + lst))
(sum '(1 2 3)) => (apply + '(1 2 3)) => (+ 1 2 3) => 6
```

Returning to our penguins

```
(define average-gentoo-mass
  (let* ([gentoos
          (filter (λ (p) (eq? (first p) 'Gentoo)) penguins)]
         [masses (map third gentoos)]
         [total-mass (apply + masses)]
         [num-gentoos (length gentoos)])
    (/ total-mass num-gentoos)))
```

Applying with some fixed arguments

`(apply proc v… lst)`

`apply` takes a variable number of arguments where the final one is a list and applies `proc` to all of those arguments

```
(apply proc 1 2 3 '(4 5 6)) => (proc 1 2 3 4 5 6)
```

Recap

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
```

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!