Abstractions and $\lambda$

22 October 2020
Design recipes for abstractions
Design recipes for abstractions: Creation
1. Find the corresponding differences.
2. Replace the differences with names and use these names as parameters to the functions.
3. Formulate the original functions in terms of the new one.

Don’t forget to pass along the new parameters in the recursive calls
Example

Analyze the similarities and differences between the following two functions. Create an abstracted function and re-write them.
(require 2htdp/image)

(define C (circle 3 "solid" "red"))
(define S (square 3 "solid" "blue"))

;;; [List-of Image] -> Image
;;; Draw all images in the list placed next to each other
(check-expect (h-cat (list C S)) (beside C S))
(define (h-cat loi)
  (cond [(empty? loi) empty-image]
        [else (beside (first loi)
                     (h-cat (rest loi)))]))

;;; [List-of String] -> String
;;; Concatenate all the strings in a given list into one string
(check-expect (collapse (list "hello" "," "computer")) "hello, computer")
(define (collapse los)
  (cond [(empty? los) ""]
        [else (string-append (first los)
                             (collapse (rest los)))]))
(require 2htdp/image)

(define C (circle 3 "solid" "red"))
(define S (square 3 "solid" "blue"))

;; [List-of Image] -> Image
;; Draw all images in the list placed next to each other
(check-expect (h-cat (list C S)) (beside C S))
(define (h-cat loi)
  (cond [(empty? loi) empty-image]
        [else
         (beside (first loi)
                 (h-cat (rest loi)))]))

;; [List-of String] -> String
;; Concatenate all the strings in a given list into one string
(check-expect (collapse (list "hello", "computer")) "hello, computer")
(define (collapse los)
  (cond [(empty? los) ""
       [else
        (string-append (first los)
                       (collapse (rest los)))]))
(require 2htdp/image)

(define C (circle 3 "solid" "red"))
(define S (square 3 "solid" "blue"))

;; [List-of Image] -> Image
;; Draw all images in the list placed next to each other
(check-expect (h-cat (list C S)) (beside C S))
(define (h-cat loi base)
  (cond [(empty? loi) base]
        [else
         (beside (first loi)
                 (h-cat (rest loi) base))]))

;; [List-of String] -> String
;; Concatenate all the strings in a given list into one string
(check-expect (collapse (list "hello" ", " "computer"))
              "hello, computer")
(define (collapse los base)
  (cond [(empty? los) base]
        [else
         (string-append (first los)
                        (collapse (rest los) base))]))
(require 2htdp/image)

(define C (circle 3 "solid" "red"))
(define S (square 3 "solid" "blue"))

;; [List-of Image] -> Image
;; Draw all images in the list placed next to each other
(check-expect (h-cat (list C S)) (beside C S))
(define (h-cat loi base comb)
  (cond [(empty? loi) base]
        [else
         (comb (first loi)
               (h-cat (rest loi) base comb))]))

;; [List-of String] -> String
;; Concatenate all the strings in a given list into one string
(check-expect (collapse (list "hello" "," "," "computer"))
              "hello, computer")
(define (collapse los base comb)
  (cond [(empty? los) base]
        [else
         (comb (first los)
               (collapse (rest los) base comb))]))
(define (combine l base comb)
  (cond [(empty? l) base]
        [else
         (comb (first l)
               (combine (rest l) base comb))])))

(define (h-cat loi)
  (combine loi empty-image beside))

(define (collapse los)
  (combine los """" string-append))
;; [List-of X] -> X
;; Use `comb` to combine the Xs one-by-one, ending
;; with `base`
(define (combine l base comb)
  (cond [(empty? l) base]
        [else
         (comb (first l)
               (combine (rest l) base comb))])))
;; [List-of X] -> X
;; Use `comb` to combine the Xs one-by-one, ending
;; with `base`
(define (combine l base comb)
  (cond [(empty? l) base]
        [else
         (comb (first l)
              (combine (rest l) base comb))])))
;; [List-of X] -> X
;; Use `comb` to combine the Xs one-by-one, ending with `base`
(define (combine l base comb)
  (cond [(empty? l) base]
        [else
         (comb (first l)
               (combine (rest l) base comb))])))
Design recipes for abstractions:

Use
When you’re writing a function and you recognize that it can be designed with an existing abstraction,

Match the signatures and purpose statements

Create a `local` template.

Design the rest of the helper function as before.

Skip explicitly testing local helper functions, but make sure you’re sufficiently testing the outer function. All code should be covered by test cases.
Are you making a single value?

Probably you can use foldr.

But if you’re asking a true–false question, consider andmap or ormap.

Are you making a list?

Consider

map (if you’re changing items) or

filter (if you’re removing items)

...or one more!
;; Natural, [Natural -> X] -> [List-of X]
;; Produce (list (f 0) ... (f (- n 1))
(define (build-list n f) ...)

> (identity 3)
3
> (build-list 3 identity)
(list 0 1 2)
> (build-list 4 sqr)
(list 0 1 4 9)
(define (new-function k l ...) 
  (local [(define (helper ...) 
              default-value)] 
    (existing-abstraction k l ... helper)))
Practice using built-in higher-order functions
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

```
(require 2htdp/image)
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

;; [List-of Image] -> [List-of Image]
;; Produce list of only those images that are wider than they are tall
(check-expect (wide-only (list I1 I2 I3 I4 I5))
              (list I2 I4))
(define (wide-only loi) ...)
```
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

```scheme
(require 2htdp/image)
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

;; [List-of Image] -> [List-of Image]
;; Produce list of only those images that are wider than they are tall
(check-expect (wide-only (list I1 I2 I3 I4 I5))
  (list I2 I4))
(define (wide-only loi)
  (local [;; Image -> Boolean
    (define (wide? img)
      (> (image-width img) (image-height img)))]
    (filter wide? loi)))
```
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

```
(require 2htdp/image)
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

;; [List-of Image] -> Boolean
;; Are all of the images in loi taller than they are wide?
(check-expect (all-tall? (list I1 I2 I3 I4 I5)) #false)
(check-expect (all-tall? (list I1 I3)) #true)
(define (all-tall? loi) ...)
```
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

(require 2htdp/image)
(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

;; [List-of Image] -> Boolean
;; Are all of the images in loi tall?
(check-expect (all-tall? (list I1 I2 I3 I4 I5)) #false)
(check-expect (all-tall? (list I1 I3)) #true)
(define (all-tall? loi)
  (local [;; Image -> Boolean
    (define (tall? img)
      (< (image-width img) (image-height img)))]
    (andmap tall? loi)))
;;; [List-of Number] -> Number
;;; Sum the elements of a list
(check-expect (sum (list 1 2 3 4)) 10)
(define (sum lon) ...
;; [List-of Number] -> Number
;; Sum the elements of a list
(check-expect (sum (list 1 2 3 4)) 10)
(define (sum lon)
  (foldr ... ... lon))
(check-expect (sum (list 1 2 3 4)) 10)
(define (sum lon)
  (foldr ... ... lon))
;; [List-of Number] -> Number
;; Sum the elements of a list
(check-expect (sum (list 1 2 3 4)) 10)
(define (sum lon)
  (foldr + 0 lon))
;;; Natural -> Natural
;;; Produce the sum of the first n natural numbers
(check-expect (sum-to 3) (+ 0 1 2))
(define (sum-to n) ...)
;; Natural -> Natural
;; Produce the sum of the first n natural numbers
(check-expect (sum-to 3) (+ 0 1 2))
(define (sum-to n)
  (foldr ... ... (build-list n ...)))
;; Natural \rightarrow Natural
;; Produce the sum of the first n natural numbers
(check-expect (sum-to 3) (+ 0 1 2))
(define (sum-to n)
  (foldr ... ... (build-list n identity)))
;; Natural -> Natural
;; Produce the sum of the first n natural numbers
(check-expect (sum-to 3) (+ 0 1 2))
(define (sum-to n)
  (foldr + 0 (build-list n identity)))
Functions and names
Consider some values we can use in Racket:

**Numbers:** 1, 17.8, 4/5

**Booleans:** #true, #false

**Lists:** '(), (cons 7 '())

...

**Function names:** less-than-5?, first-is-apple?

given

(define (less-than-5? n) ...)
(define (first-is-apple? a b) ...)
Consider some values we can use in Racket:

**Numbers**: 1, 17.8, 4/5

**Booleans**: #true, #false

**Lists**: '(), (cons 7 '())

... 

**Function names**: less-than-5?, first-is-apple?

given

```racket
(define (less-than-5? n) ...)
(define (first-is-apple? a b) ...)
```

Why do only function values require names?
Consider this higher-order function:

```scheme
(define (choose a b p)
  (cond [(p a) a]
        [(p b) b]
        [else #false]))
```

;;; X, X, [X -> Boolean] -> [Maybe X]
;;; Choose one of the inputs satisfied by
;;; pred. p or #false if neither is
We can use it like this, using a local for the predicate:

```scheme
(local [(define (first-is-apple? l)
         (string=? (first l) "apple"))]
       (choose (list "apple" "banana")
               (list "cherry" "cherry"
                   first-is-apple?))

But if we needed to name every kind of value, it would be painful:

```scheme
(local [(define (first-is-apple? l)
         (string=? (first l) "apple"))]
       (define a1 (list "apple" "banana"))
       (define b1 (list "cherry" "cherry"))
       (choose a1 b1 first-is-apple?))
```
So, can we avoid naming functions?

Instead of writing

```
(local [(define (first-is-apple? l)
         (string=? (first l) "apple"))]
    ... first-is-apple? ...))
```

we’d like to write

```
... function that takes l and produces (string=? (first l) "apple") ...
```

skipping the `local` expression since we’re only using this function once.
We can do this … in the *Intermediate Student with lambda* language.
**Lambda expressions** are special forms that evaluate to procedures.
The meaning of a $\lambda$ expression

$$(\lambda (x) (* x x))$$

A procedure

Of one argument, “x”

That returns the square of x
The meaning of a $\lambda$ expression

$$(\lambda \ (x \ y) \ (/ \ (+ \ x \ y) \ 2))$$

A procedure

Of two arguments, “x” and “y”

That returns the average of x and y
Calling this keyword λ (or lambda) is a convention derived from the Lambda Calculus, developed by Alonzo Church in the 1930s, where λ is used to mean “function”.

Church’s choice of this notation was arbitrary, but arbitrary choices can have long-lasting consequences. For another example, consider why programmers use “foo” and “bar” as placeholder names.
An example of an anonymous function value is

\[
(\lambda \ (a \ b) \ (\text{string}=? \ (\text{first} \ l) \ "\text{apple}"))
\]

So, using \( \lambda \), the original example becomes

\[
(\text{choose} \ (\text{list} \ "\text{apple}" \ "\text{banana}") \\
(\text{list} \ "\text{cherry}" \ "\text{cherry}")) \\
(\lambda \ (l) \\
\quad (\text{symbol}=? \ (\text{first} \ l) \ "\text{apple}")))
\]
The function position of an *application* (i.e., a function call) no longer needs to be a name, e.g.,

\[
> \ (\text{define} \ (\text{add10} \ x) \ (+ \ 10 \ x))
> \ (\text{add10} \ 17)
27
\]

It can now be the anonymous function value:

\[
> \ ((\lambda \ (x) \ (+ \ x \ 10)) \ 17)
27
\]
Positional association

\[(\lambda(x\ y)\ (\div\ (+\ x\ y)\ 2))\ 1066\ 2019\]  

Which actual argument is substituted for \(x\)?  
The one in the first position, since \(x\) is first.

Which actual argument is substituted for \(y\)?  
The one in the second position, since \(y\) is second.
So, what’s the difference between

\[
\begin{align*}
(\text{define } (f \ a \ b) \\
(\ + \ a \ b))
\end{align*}
\]

and

\[
\begin{align*}
(\text{define } f \\
(\lambda (a \ b) \\
(\ + \ a \ b)))
\end{align*}
\]
So, what’s the difference between

\[
\text{(define (f a b)}
\text{ (+ a b))}
\]

and

\[
\text{(define f}
\text{ (λ (a b)}
\text{ (+ a b)))}
\]

\?

\text{Nothing! The former is just a shorthand (“syntactic sugar”) for the latter.}
Design with $\lambda$
Anonymous functions are great for when we want to pass a function as an argument to another function.

Just as we can call a function on 5 rather than write (define N 5) and call it on N, we can now call a function on another function value without first defining it.
Anonymous functions work great with `filter`, `map`, etc.:

```scheme
(define (eat-apples l)
  (filter (λ (a)
            (not (string=? a "apple")))
         l))

(define (inflate-by-4% l)
  (map (λ (n) (* n 1.04)) l))

(define (total-blue l)
  (foldr (λ (c n)
          (+ (color-blue c) n))
         0 1))
```
With $\lambda$, we can update the design recipe for abstractions to skip having a local definition.
Example: [lambda.rkt]
Remember that we can have functions that return other functions.

We did this using **local** to give a name to the function and then returning that name.

But with **lambda**, we can simplify it and just have a function whose body is the **lambda** expression (anonymous function) it returns.
Suppose we need to filter different strings:

\[
\begin{align*}
& (\text{filter } (\lambda (a) (\text{string=} a \ "apple") ) \ l) \\
& (\text{filter } (\lambda (a) (\text{string=} a \ "banana") ) \ l) \\
& (\text{filter } (\lambda (a) (\text{string=} a \ "cherry") ) \ l)
\end{align*}
\]

Instead of repeating this whole lambda expression, we can abstract:

\[
\text{;; } \text{String } \rightarrow \ [\text{String } \rightarrow \ \text{[Boolean]}}
\]

\[
\begin{align*}
& (\text{define } (\text{mk-is-str } s) \\
& \quad (\lambda (a) (\text{string=} s a)))
\end{align*}
\]

\[
\begin{align*}
& (\text{filter } (\text{mk-is-str } "apple") \ l) \\
& (\text{filter } (\text{mk-is-str } "banana") \ l) \\
& (\text{filter } (\text{mk-is-str } "cherry") \ l)
\end{align*}
\]
Suppose we need to filter different strings:

\[
\begin{align*}
&\text{(filter (λ (a) (string=? a "apple")) l)} \\
&\text{(filter (λ (a) (string=? a "banana")) l)} \\
&\text{(filter (λ (a) (string=? a "cherry")) l)}
\end{align*}
\]

Instead of repeating this whole lambda expression, we can abstract:

\[
\text{;; String -> [String -> Boolean]}
\]

\[
\text{(define (mk-is-str s)}
\]

\[
\text{\hspace{10mm} (λ (a) (string=? s a)))}
\]

\[
\text{(filter (mk-is-str "apple") l)} \\
\text{(filter (mk-is-str "banana") l)} \\
\text{(filter (mk-is-str "cherry") l)}
\]

\text{mk-is-str is a "curried" version of string=?}
Revisiting the `negate` function (a function that produces the negation of a function using `local`):

```scheme
;;; negate : [X -> Boolean] -> [X -> Boolean]
;;; Create a new function that does the negation
;;; of the given function
(check-expect ((negate even?) 2) #false)
(check-expect ((negate even?) 3) #true)
(define (negate p)
  (λ (x) (not (p x)))))
```
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith
Laney Strange