CMPU 101 § 3 · Problem-Solving and Abstraction

Sorting a List

6 October 2020
Programs are collections of definitions:

- Data definitions
- Structure type definitions
- Constant definitions
- Function definitions
Programs are collections of definitions:

Data definitions

;;; A HogwartsHouse is
;;; (make-hogwarts-house String String String)
;;; Interp.: (make-hogwarts-house name color animal)
;;; is a representation of a Hogwarts house with
;;; name is the house name
;;; color is the primary house color
;;; animal is the name of the house mascot

Structure type definitions

Constant definitions

Function definitions
Programs are collections of definitions:

**Data definitions**

```
;; A HogwartsHouse is
;;   (make-hogwarts-house String String String)
;; Interp.: (make-hogwarts-house name color animal)
;; is a representation of a Hogwarts house with
;;   name is the house name
;;   color is the primary house color
;;   animal is the name of the house mascot
```

**Structure type definitions**

Data definitions are for **people**, not for DrRacket. They tell you or other readers what type of values your functions should expect and how to interpret those values.
Programs are collections of definitions:

Data definitions

Structure type definitions

(define-struct hogwarts-house [name color animal])

Constant definitions

Function definitions
Programs are collections of definitions:

Data definitions

Structure type definitions

(define-struct hogwarts-house [name color animal])

Constant definitions

Function definitions

Structure definitions are for DrRacket. They tell Racket to define helpful functions for working with compound data, namely a constructor, a type predicate, and selectors for each of the fields.
Programs are collections of definitions:

Data definitions

Structure type definitions

Constant definitions

(define NUM-HOUSES 4)
(define H (make-hogwarts-house "Hufflepuff" "yellow" "badger"))

Function definitions
Programs are collections of definitions:

Data definitions

Structure type definitions

Constant definitions

(define NUM-HOUSES 4)
(define H (make-hogwarts-house "Hufflepuff" "yellow" "badger"))

Function definitions

*Constant definitions give names to values you want to re-use in your code, either to make the functions work or for testing.*
Programs are collections of definitions:

Data definitions

Structure type definitions

Constant definitions

Function definitions

;; summon : String -> String
;; Perform a summoning spell for s
(check-expect (summon "example") "Accio example!")
(define (summon s)
  (string-append "Accio " s "!"))
Programs are collections of definitions:

Data definitions

Structure type definitions

Constant definitions

Function definitions

;; summon : String -> String
;; Perform a summoning spell for s
(check-expect (summon "example") "Accio example!")
(define (summon s)
  (string-append "Accio " s "!"))

*Functions define actions the computer can take for any number of possible inputs.*
Last class, we saw data definitions that used

*Self-reference*: A data type defined in terms of itself
*Reference*: Data defined in terms of another data type

And, correspondingly, functions with

*Natural recursion*: A function that does some work and then calls itself on a smaller piece of the input to do the rest
*Natural helpers*: A function that calls another function to deal with the details of another type of data
We want to divide labor among functions, not have long, complicated functions that are difficult to read or test.

Design one template per data definition.

Formulate auxiliary function definitions when one data definition points to a second data definition.
When we’re writing a function and realize another function is needed to deal with a subproblem, we don’t want to forget about it, but we also don’t want to abandon the function we’re writing.

Instead, we add the new function to our wish list.

To do this, write a complete function header (signature, purpose, and stub).

This lets you test other parts of the program that have finished, even though many of those tests will fail.
Working with Lists
Recall the self-referential data definition for a list of numbers:

;;;; A ListOfNumbers is one of:
;;;; - '()
;;;; - (cons Number ListOfNumbers)
(define LON1 '())
(define LON2 (cons 60 (cons 42 empty)))
;;
(define (lon-template lon)
  (cond [(empty? lon) (...)]
        [(cons? lon)
           (... (first lon)
                (fn-for-lon (rest lon)))]))
Recall the self-referential data definition for a list of numbers:

```
;; A ListOfNumbers is one of:
;;   - '()
;;   - (cons Number ListOfNumbers)
(define LON1 '())
(define LON2 (cons 60 (cons 42 empty)))#
(define (lon-template lon)
  (cond [(empty? lon) (...)]
        [(cons? lon)
          (... (first lon)
               (fn-for-lon (rest lon)))]))
```
Recall the self-referential data definition for a list of numbers:

;;;; A ListOfNumbers is one of:
;;;; - '()
;;;; - (cons Number ListOfNumbers)
(define LON1 '())
(define LON2 (cons 60 (cons 42 empty)))
#
(define (lon-template lon)
  (cond [(empty? lon) (...)]
        [(cons? lon)
         (... (first lon)
              (fn-for-lon (rest lon)))]))
Recall the self-referential data definition for a list of numbers:

```
;;; A ListOfNumbers is one of:
;;; - '()
;;; - (cons Number ListOfNumbers)
(define LON1 '())
(define LON2 (cons 60 (cons 42 empty)))

(define (lon-template lon)
  (cond [(empty? lon) (...)]
     [(cons? lon)
       (... (first lon)
       (fn-for-lon (rest lon)))]))
```

Examples we can use for testing

A template for a function that consumes a list of numbers
A ListOfNumbers is one of:

- '
- (cons Number ListOfNumbers)

(define LON1 '())
(define LON2 (cons 60 (cons 42 empty)))

(define (lon-template lon)
  (cond [(empty? lon) (...)]
        [(cons? lon)
         (... (first lon)
              (lon-template (rest lon)))]))

Use this definition to design a function that doubles every number in a list of numbers.
;;; double-all : ListOfNumbers -> ListOfNumbers
;;; Double every number in the given list
(check-expect (double-all empty) empty)
(check-expect (double-all LON2)
  (cons 120 (cons 84 empty)))
(check-expect (double-all (cons 10 (cons 20 (cons 50 empty))))
  (cons 20 (cons 40 (cons 100 empty))))
(define (double-all lon)
  (cond [(empty? lon) '()]
        [else
         (cons (* 2 (first lon))
               (double-all (rest lon)))]))
Example: Arranging images
You have a bunch of pictures you’d like to store as data and present in an orderly way.

1. Design a data definition to represent an arbitrary number of images.

2. Design a function `arrange-images` to consume an arbitrary number of images and arrange them left-to-right in increasing order of size.
You have a bunch of pictures you’d like to store as data and present in an orderly way.

1. Design a data definition to represent an arbitrary number of images.

2. Design a function `arrange-images` to consume an arbitrary number of images and arrange them left-to-right in increasing order of size.
;; Data definitions
;;

;; A ListOfImages is one of:
;; - '()
;; - (cons Image ListOfImages)
;; interp. An arbitrary number of images
(define LOI1 '())
(define LOI2 (cons (rectangle 10 20 "solid" "blue")
                   (cons (rectangle 20 30 "solid" "red")
                       '()))

;;
(define (loi-template loi)
  (cond [(empty? loi) ( ... )]
        [(cons? loi) ( ... (first loi)
                           ( ... (loi-template (rest loi))))])))
You have a bunch of pictures you’d like to store as data and present in an orderly way.

1. Design a data definition to represent an arbitrary number of images.

2. Design a function `arrange-images` to consume an arbitrary number of images and arrange them left-to-right in increasing order of size.
;; Functions
;;

;; arrange-images : ListOfImages -> Image
;; Lay out images left to right in increasing order of size
(check-expect (arrange-images '()) empty-image)
(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
        (cons (rectangle 20 30 "solid" "red")
              '()))
  (beside (rectangle 10 20 "solid" "blue")
           (rectangle 20 30 "solid" "red")
           empty-image))

Write signature, purpose, and examples
;; Functions

;; arrange-images : ListOfImages -> Image
;; Lay out images left to right in increasing order of size
(check-expect (arrange-images '()) empty-image)
(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
        (cons (rectangle 20 30 "solid" "red")
              '()))
  (beside (rectangle 10 20 "solid" "blue")
           (rectangle 20 30 "solid" "red")
           empty-image))

Why have empty-image here? It’s invisible.
;;
;; Functions
;;

;; arrange-images : ListOfImages -> Image
;; Lay out images left to right in increasing order of size
(check-expect (arrange-images '()) empty-image)
(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
       (cons (rectangle 20 30 "solid" "red") '()))
  (beside (rectangle 10 20 "solid" "blue")
          (rectangle 20 30 "solid" "red")
          empty-image)))

These aren’t just test cases; they’re examples of how the function should work. The empty-image is built from '() in the input, then we add the other two images.
Functions

; arrange-images : ListOfImages -> Image
; Lay out images left to right in increasing order of size
(check-expect (arrange-images '()) empty-image)
(check-expect (arrange-images
   (cons (rectangle 10 20 "solid" "blue"
     (cons (rectangle 20 30 "solid" "red"
       '()))
   (beside (rectangle 10 20 "solid" "blue"
     (rectangle 20 30 "solid" "red"
       empty-image)))))
(define arrange-images (loi)
  (cond [(empty? loi) (...)]
        [(cons? loi)
         (... (first loi)
             (arrange-images (rest loi))))])
When we try to fill out the template, it becomes clear that we can’t lay out any of the images until we’ve sorted them by size.

This is a *function composition* problem because the function needs to perform more than one distinct operation on its input.

So, we replace the template with the distinct operations as separate functions.
;; Functions
;;

;; arrange-images : ListOfImages -> Image
;; Lay out images left to right in increasing order of size
(define (arrange-images loi)
  (layout-images (sort-images loi)))

;; Wish list:

;; layout-images : ListOfImages -> Image
;; Place images besides each other in order of list

;; sort-images : ListOfImages -> ListOfImages
;; Sort images in increasing order of size.
Now that we’ve identified that *arrange-images* is a composition of two functions, we should reconsider our test cases.

The two helpers, *layout-images* and *sort-images* should be tested separately. For *arrange-images* we want to check that the composition is right.
We don’t need to test the base case since `arrange-images` isn’t recursive.

(check-expect (arrange-images '()) empty-image)
(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
       (cons (rectangle 20 30 "solid" "red")
             '()))
  (beside (rectangle 10 20 "solid" "blue")
           (rectangle 20 30 "solid" "red")
           empty-image))
But this test case isn’t adequate. It would be satisfied if `arrange-images` only called `layout-images` without sorting.

(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
    (cons (rectangle 20 30 "solid" "red")
      '()))
  (beside (rectangle 10 20 "solid" "blue")
    (rectangle 20 30 "solid" "red")
    empty-image))
(check-expect (arrange-images
  (cons (rectangle 10 20 "solid" "blue")
    (cons (rectangle 20 30 "solid" "red")
      '()))))
(beside (rectangle 10 20 "solid" "blue")
  (rectangle 20 30 "solid" "red")
  empty-image))

(check-expect (arrange-images
  (cons (rectangle 20 30 "solid" "red")
    (cons (rectangle 10 20 "solid" "blue")
      '()))))
(beside (rectangle 10 20 "solid" "blue")
  (rectangle 20 30 "solid" "red")
  empty-image))

Better!
;; layout-images : ListOfImages → Image
;; Place images besides each other in order of list
;; layout-images : ListOfImages -> Image
;; Place images besides each other in order of list
(check-expect (layout-images '()) empty-image)
(check-expect (layout-images
  (cons (rectangle 10 20 "solid" "blue")
    (cons (rectangle 20 30 "solid" "red")
      '()))
  (beside (rectangle 10 20 "solid" "blue")
    (rectangle 20 30 "solid" "red")
    empty-image))
;; layout-images : ListOfImages -> Image
;; Place images besides each other in order of list
(check-expect (layout-images '()) empty-image)
(check-expect (layout-images
    (cons (rectangle 10 20 "solid" "blue")
    (cons (rectangle 20 30 "solid" "red")
    '())))
    (beside (rectangle 10 20 "solid" "blue")
    (rectangle 20 30 "solid" "red")
    empty-image))

(define (layout-images loi)
  (cond [(empty? loi) (...)]
    [(cons? loi)
      (... (first loi)
        (layout-images (rest loi)))]))
;; layout-images : ListOfImages -> Image
;; Place images besides each other in order of list
(check-expect (layout-images '()) empty-image)
(check-expect (layout-images
    (cons (rectangle 10 20 "solid" "blue")
        (cons (rectangle 20 30 "solid" "red")
            '()))
    (beside (rectangle 10 20 "solid" "blue")
        (rectangle 20 30 "solid" "red")
        empty-image))

(define (layout-images loi)
    (cond [(empty? loi) empty-image]
        [(cons? loi)
            (beside (first loi)
                (layout-images (rest loi)))]))
sort-images : ListOfImages → ListOfImages
;; Sort images in increasing order of size.
;; sort-images : ListOfImages -> ListOfImages
;; Sort images in increasing order of size.
(check-expect (sort-images '()) '())
(check-expect
  (sort-images (cons (rectangle 10 20 "solid" "blue")
                 (cons (rectangle 20 30 "solid" "red")
                       '())))
  (cons (rectangle 10 20 "solid" "blue")
        (cons (rectangle 20 30 "solid" "red")
               '())))
(check-expect
  (sort-images (cons (rectangle 20 30 "solid" "red")
                 (cons (rectangle 10 20 "solid" "blue")
                       '())))
  (cons (rectangle 10 20 "solid" "blue")
        (cons (rectangle 20 30 "solid" "red")
               '()))))
;; sort-images : ListOfImages -> ListOfImages
;; Sort images in increasing order of size.
(check-expect (sort-images '()) '())
...

(define (sort-images loi)
  (cond [(empty? loi) (...)]
        [(cons? loi)
         (... (first loi)
              (sort-images (rest loi)))]))
;;; sort-images : ListOfImages -> ListOfImages
;;; Sort images in increasing order of size.
(check-expect (sort-images '()) '())
...
(define (sort-images loi)
  (cond [(empty? loi) '()] [(cons? loi)
        (... (first loi)
             (sort-images (rest loi)))]))
Sometimes the first element of the list goes right at the beginning; it stays put.

Sometimes it doesn’t. If our test lists were longer, we might find that the first element in the list goes somewhere in the middle of the result.

Let’s add a longer test case!
;; sort-images : ListOfImages -> ListOfImages
;; Sort images in increasing order of size.
(check-expect (sort-images '()) '())
...
(check-expect
  (sort-images
   (cons (rectangle 30 40 "solid" "green")
       (cons (rectangle 10 20 "solid" "blue")
           (cons (rectangle 20 30 "solid" "red")
               '()))))

(define (sort-images loi)
  (cond
   [(empty? loi) '()]
   [(cons? loi)
     [(cons? loi)
       (... (first loi)
           (sort-images (rest loi)))])))
We trust that the result of the natural recursion – the call to sort-images on the rest of the list of images – will work; the result will be sorted.
(define (sort-images loi)
  (cond [(empty? loi) '()][(cons? loi)
      ([... (first loi)
         (sort-images (rest loi)))]))

We want to put the first element somewhere in the sorted list, but we can't tell where without going through the sorted list.
Whenever we need to operate on arbitrary-sized data we should use a function to do it, so we want a new helper function, `insert`.

```scheme
(define (sort-images loi)
  (cond [(empty? loi) '()] ; Check if the list is empty
        [(cons? loi)
          ; Insert the first element of the list
          [insert (first loi)
            (sort-images (rest loi))]]))
```
;; sort-images : ListOfImages -> ListOfImages
;; Sort images in increasing order of size.
...
(define (sort-images loi)
  (cond [(empty? loi) '()]
        [(cons? loi)
         (insert (first loi)
                 (sort-images (rest loi)))]))
...

;; Wish list

;; insert : Image ListOfImages -> ListOfImages
;; Insert img in proper place in loi (in increasing order of size).
;; Assume loi is already sorted!
(define (insert img loi) ...)
;; insert : Image, ListOfImages -> ListOfImages
;; Insert img in proper place in loi (in increasing order of size).
;; Assume loi is already sorted!
(define (insert img loi) ...)
We need to define tests for insert, but it’s painful to keep repeating these images, so let’s define some constants

;; Constants

(define I1 (rectangle 10 20 "solid" "blue"))
(define I2 (rectangle 20 30 "solid" "red"))
(define I3 (rectangle 30 40 "solid" "green"))

...
;; Constants

(define I1 (rectangle 10 20 "solid" "blue"))
(define I2 (rectangle 20 30 "solid" "red"))
(define I3 (rectangle 30 40 "solid" "green"))

...
;; insert : Image, ListOfImages -> ListOfImages
;; Insert img in proper place in loi (in increasing order
;; of size).
;; Assume loi is already sorted!
(define (insert img loi)
  (cond [(empty? loi)
         (... img)]
       [(cons? loi)
        (... img
           (first loi)
           (insert (... img)
                     (rest loi))))])

The template is slightly different because we have **both** a ListOfImages and an Image as inputs.
```scheme
(define (insert img loi)
  (cond [(empty? loi)
         (cons img '())
       [(cons? loi)
         (... img
             (first loi)
             (insert (... img)
                       (rest loi)))])
  
Base case: If the list is empty, there's only one place to put `img`.)
```
For the natural recursion case:

If \texttt{img} isn’t larger than the first image in \texttt{loi}, then put it at the start.

Otherwise, \texttt{img} goes somewhere later in the list; keep looking for the right spot.

So, we need to be able to compare the sizes of two images.
;; insert : Image, ListOfImages -> ListOfImages
;; Insert img in proper place in loi (in increasing order
;; of size).
;; Assume loi is already sorted!
(define (insert img loi)
  (cond [(empty? loi)
         (cons img '())
    [(cons? loi)
     (if (larger? img (first loi))
       (... img
       (first loi)
       (insert (... img)
       (rest loi)))
     (... img
     (first loi)
     (insert (... img)
     (rest loi)))]))

This is a function about sorting, not about comparing image areas. That’s a different problem, so we want it to be a new function, which we can call larger.
;; insert : Image, ListOfImages → ListOfImages
;; Insert img in proper place in loi (in increasing order
;; of size).
;; Assume loi is already sorted!
(define (insert img loi)
  (cond [(empty? loi)
          (cons img '())
        [(cons? loi)
          (if (larger? img (first loi))
            (cons (first loi)
                  (insert img) (rest loi))
            (cons img loi))])))
;;; insert : Image, ListOfImages → ListOfImages
;;; Insert img in proper place in loi (in increasing order 
;;; of size).
;;; Assume loi is already sorted!
(define (insert img loi)
  (cond [[(empty? loi)
      (cons img '())
    [(cons? loi)
      (if (larger? img (first loi))
        (cons (first loi)
          (insert img (rest loi)))
        (cons img loi))]]))

;;; Wish list

;;; Image Image → Boolean
;;; Produce #true if img1 is larger than img2 (by area)
(define (larger? img1 img2) ...)
;; larger? : Image, Image -> Boolean
;; Produce #true if img1 is larger than img2 (by area)
(define (larger? img1 img2) ...)
;; larger? : Image, Image -> Boolean
;; Produce #true if img1 is larger than img2 (by area)
(check-expect (larger? (rectangle 3 4 "solid" "red")
                    (rectangle 2 6 "solid" "red"))
            #false)
(check-expect (larger? (rectangle 5 4 "solid" "red")
                    (rectangle 2 6 "solid" "red"))
            #true)
(check-expect (larger? (rectangle 3 5 "solid" "red")
                    (rectangle 2 6 "solid" "red"))
            #true)
(check-expect (larger? (rectangle 3 4 "solid" "red")
                    (rectangle 5 6 "solid" "red"))
            #false)
(check-expect (larger? (rectangle 3 4 "solid" "red")
                    (rectangle 2 7 "solid" "red"))
            #false)
(define (larger? img1 img2) ...)
;; larger? : Image, Image -> Boolean
;; Produce #true if img1 is larger than img2 (by area)
(check-expect (larger? (rectangle 3 4 "solid" "red")
                (rectangle 2 6 "solid" "red"))
       #false)

...(define (larger? img1 img2)
   (... img1 img2))
;; larger? : Image, Image -> Boolean
;; Produce #true if img1 is larger than img2 (by area)
(check-expect (larger? (rectangle 3 4 "solid" "red"))
              (rectangle 2 6 "solid" "red"))
    #false

... 

(define (larger? img1 img2)
  (> (* (image-width img1) (image-height img1))
     (* (image-width img2) (image-height img2))))
arrange-images

sort-images

insert

layout-images ✔

larger? ✔
arrange-images

- sort-images
- layout-images
- insert
  - larger?
arrange-images ✔
  └── sort-images ✔
    └── insert ✔
        └── larger? ✔
  └── layout-images ✔
Acknowledgments

This lecture incorporates material from:

- Matthias Felleisen
- Robert Bruce Findler
- Matthew Flatt
- Gregor Kiczales
- Shriram Krishnamurthi
- Marc Smith