Data So Far

• Built-in atomic data: `num`, `bool`, `string`, and `image`

• Built-in compound data: `posn`

• Programmer-defined compound data: `define-struct` plus a data definition

• Programmer-defined data with varieties: data definition with “either”

Today: more examples
Example 1: Managing Grades

Suppose that we need to manage exam grades
Example 1: Managing Grades

Suppose that we need to manage exam grades

100
Example 1: Managing Grades

Suppose that we need to manage exam grades
Example 1: Managing Grades

Suppose that we need to manage exam grades

100  0  \( R_x \)
Example 1: Managing Grades

Suppose that we need to manage exam grades

- Record a grade for each student
- Distinguish zero grade from missing the exam

We want to implement passed-exam?
Programming with Grades

Data

• Use a number for a grade, obviously
Programming with Grades

Data

• Use a number for a grade, obviously

• For a non-grade, use the built-in constant ' ()

' () is something that you can use to represent nothing.

It's not a num, bool, string, image, or posn.
Programming with Grades

Data

; A grade is either
; - num
; - '()'
Programming with Grades

Data

; A grade is either
; - num
; - '()' 

Examples:

100
0
'()'
Programming with Grades

Signature, Purpose, and Header

; passed-exam? : grade -> bool
Programming with Grades

Signature, Purpose, and Header

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
Programming with Grades

Signature, Purpose, and Header

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  ...)

Programming with Grades

Examples

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  ...
)

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? 0) #false)
(check-expect (passed-exam? '()) #false)
Programming with Grades

Template

; passed-exam? : grade → bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  (cond
   [(number? g) ...]
   [(empty? g) ...]))

varieties ⇒ cond

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? 0) #false)
(check-expect (passed-exam? '()) #false)
Programming with Grades

Body

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
;
; (define (passed-exam? g)
  (cond
   [(number? g) ...]
   [(empty? g) ...]))

(define (passed-exam? g)
  (cond
   [(number? g) (>= g 70)]
   [(empty? g) #false]))

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? 0) #false)
(check-expect (passed-exam? '() ) #false)
Grades and Re-takes

Suppose that we allow one re-test per student

100

0 80

Rx
Grades and Re-takes

Suppose that we allow one re-test per student

; A grade is either
;   - num
;   - posn
;   - '()'
Programming with Grades and Retests

Signature, Purpose, and Header

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  ...)

Examples

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
    ...)

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? (make-posn 0 80)) #true)
(check-expect (passed-exam? '()) #false)
; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  (cond
   [(number? g) ...]
   [(posn? g) ...]
   [(empty? g) ...]))

varieties ⇒ cond

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? (make-posn 0 80)) #true)
(check-expect (passed-exam? '()) #false)
Programming with Grades and Retests

Template

; passed-exam? : grade -> bool
; Determines whether g is 70 or better
(define (passed-exam? g)
  (cond
    [(number? g) ...]
    [(posn? g) ... (posn-passed-exam? g) ...]
    [(empty? g) ...]))

(check-expect (passed-exam? 100) #true)
(check-expect (passed-exam? (make-posn 0 80)) #true)
(check-expect (passed-exam? '()) #false)
Complete Function

; passed-exam? : grade -> bool
(define (passed-exam? g)
  (cond
    [(number? g) (>= g 70)]
    [(posn? g) (posn-passed-exam? g)]
    [(empty? g) #false]))

; posn-passed-exam? : posn -> bool
(define (posn-passed-exam? p)
  (or (>= (posn-x p) 70)
      (>= (posn-y p) 70)))

Plus tests and templates...
Shapes of Data and Functions

As always, the shape of the function matches the shape of the data

; A grade is either
;  - num
;  - posn
;  - '()

; A posn is
;  (make-posn num num)

(define (func-for-grade g)
  (cond
    [(number? g) ...]
    [(posn? g) ... (func-for-posn g) ...]
    [(empty? g) ...])

(define (func-for-posn p)
  ... (posn-x p) ... (posn-y p) ..)
Summary

Today’s examples show:

• A data definition with variants need not involve structure choices

• A data definition with variants can include `make-something` directly

  ... usually when the structure by itself isn’t useful

• Implementation shape still matches the data shape

No recipe changes!