Data So Far

• Built-in atomic data: \texttt{num, bool, sym,} and \texttt{image}

• Built-in compound data: \texttt{posn}

• Programmer-defined compound data: \texttt{define-struct} plus a data definition

• Programmer-defined data with varieties: data definition with "either"

\textbf{Today:} more examples
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

• For a non-grade, use the built-in constant `empty`

`empty` is something that you can use to represent nothing.

It’s not a `num, bool, sym, image, or posn`. 
Programming with Grades

Data

; A grade is either
;   - num
;   - empty

Examples:

100
0
empty
Programming with Grades

Contract, Purpose, and Header

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

**Contract, Purpose, and Header**

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

Contract, 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? empty) 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? empty) 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? empty) false)
Grades and Re-takes

Suppose that we allow one re-test per student

; A grade is either
; - num
; - posn
; - empty
Programming with Grades and Retests

Contract, Purpose, and Header

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

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? empty) false)
Programming with Grades and Retests

**Template**

```scheme
; 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? empty) 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) ...]))

data-defn reference ⇒ template reference

(check-expect (passed-exam? 100) true)
(check-expect (passed-exam? (make-posn 0 80)) true)
(check-expect (passed-exam? empty) 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
;   - empty

; 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) ..)
Example #2: Day Planning

Suppose that we need to manage day-planner entries

<table>
<thead>
<tr>
<th>@lab</th>
<th>@office</th>
</tr>
</thead>
</table>

Each day-plan is either empty or an appointment with person and place

Implement close-blinds?

for Adam’s sensitive eyes during office meetings
Programming with Day-Plans

Data

; An day-plan is either
;   - empty
;   - (make-appt image sym)
(define-struct appt (who where))

Examples:

empty

(make-appt 'office)
Programming with Day-Plans

Contract, Purpose, and Header

; close-blinds? : day-plan -> bool
Programming with Day-Plans

Contract, Purpose, and Header

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
Programming with Day-Plans

**Contract, Purpose, and Header**

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
(define (close-blinds? dp)
  ...)


Programming with Day-Plans

Examples

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
(define (close-blinds? dp)
  ...
)(check-expect (close-blinds? empty) false)

(check-expect (close-blinds? (make-appt
  true)
  "office")

(check-expect (close-blinds? (make-appt
  false)
  "lab")
Programming with Day-Plans

Template

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
(define (close-blinds? dp)
  ...)

; An day-plan is either
;  - empty
;  - (make-appt image sym)
Programming with Day-Plans

Template

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
(define (close-blinds? dp)
  (cond
   [(empty? dp) ...]
   [(appt? dp) ...]]))

varieties ⇒ cond

; An day-plan is either
; − empty
; − (make-appt image sym)
Programming with Day-Plans

Template

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting
; with Adam at office
(define (close-blinds? dp)
  (cond
   [(empty? dp) ...]
   [(appt? dp)
     ... (appt-who dp)
     ... (appt-where dp) ...]])

compound data ⇒ extract parts

; An day-plan is either
; - empty
; - (make-appt image sym)
Programming with Day-Plans

Body

; close-blinds? : day-plan -> bool
; Determines whether dp is a meeting with Adam at office
(define (close-blinds? dp)
  (cond
   [(empty? dp) false]
   [(appt? dp)
    (and
     (image=? (appt-who dp) )
     (symbol=? (appt-where dp) 'office))])))
Shapes of Data and Functions

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

; An day-plan is either
;   - empty
;   - (make-appt image sym)

(define (close-blinds? dp)
  (cond
   [(empty? dp) ...]
   [(appt? dp)
    ... (appt-who dp)
    ... (appt-where dp) ...]]))
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!