Recall: Programming with Rabbits
Suppose we want to represent rabbits (maybe virtual rabbits, maybe real rabbits available for adoption):

- name
- weight
- favorite food

What kind of data is appropriate for a 🐇?
Suppose we want to represent rabbits (maybe virtual rabbits, maybe real rabbits available for adoption):

- name
- weight
- favorite food

What kind of data is appropriate for a 🐰?

Not *Number, Boolean, String, Image, or Posn*…
Data definitions and define-struct

We can tell DrRacket about rabbits:

```
(define-struct rabbit [name weight food])
```

which creates the following:

- `make-rabbit` (constructor)
- `rabbit-name`, `rabbit-weight`, `rabbit-food` (selectors)
- `rabbit?` (predicate)
(define-struct rabbit [name weight food])

;; A Rabbit is a (make-rabbit String PositiveNumber String)
;; interp. (make-rabbit name weight food) is a rabbit with
;;   name is the rabbit's adorable name
;;   weight is the latest weight in lbs
;;   food is the rabbit's favorite food

;; Examples:
(define FIVER (make-rabbit "Fiver" 1.6 "spinach"))
(define HAZEL (make-rabbit "Hazel" 2.5 "carrots"))
(define PETER (make-rabbit "Peter" 4 "cabbage"))

#;
(define (rabbit-temp r)
  (... (rabbit-name r)
       (rabbit-weight r)
       (rabbit-food r)))
;; feed-rabbit : Rabbit -> Rabbit
;; Feed a rabbit 0.25 lbs of food

(check-expect (feed-rabbit FIVER)
  (make-rabbit "Fiver" 1.85 "spinach"))

(check-expect (feed-rabbit HAZEL)
  (make-rabbit "Hazel" 2.75 "carrots"))

(define (feed-rabbit r)
  (make-rabbit (rabbit-name r)
    (+ 0.25 (rabbit-weight r))
    (rabbit-food r)))
Expanding our zoo: Armadillos
Programming with armadillos

Pick a representation for armadillos ("dillo" for short), where a dillo has a weight and may or may not be alive.
(define-struct dillo [weight alive?] )
;; A Dillo is (make-dillo PositiveNumber Boolean)
;; Interpretation: (make-dillo weight alive?) is an
;; armadillo with
;; weight is the weight in lbs
;; alive? is #true if the dillo is alive; #false
;; otherwise
;; Examples:
(define LIVING-DILLO (make-dillo 3.8 #true))
(define DEAD-DILLO (make-dillo 4 #false))
#
(define (dillo-temp d)
  (... (dillo-weight d) (dillo-alive? d)))
Programming with armadillos

Pick a representation for armadillos ("dillo" for short), where a dillo has a weight and may or may not be alive.

Implement \texttt{run-over-with-car}, which takes a dillo and returns a dead dillo of equal weight.
;; run-over-with-car : Dillo -> Dillo
;;   To (sadly) kill the given armadillo

(check-expect (run-over-with-car LIVING-DILLO)
  (make-dillo 3.8 #false))

(check-expect (run-over-with-car DEAD-DILLO)
  DEAD-DILLO)
;; run-over-with-car : Dillo -> Dillo
;; To (sadly) kill the given armadillo

(check-expect (run-over-with-car LIVING-DILLO)
  (make-dillo 3.8 #false))
(check-expect (run-over-with-car DEAD-DILLO)
  DEAD-DILLO)

(define (run-over-with-car d)
  (... (dillo-weight d) (dillo-alive? d)))
;; run-over-with-car : Dillo -> Dillo
;;    To (sadly) kill the given armadillo

(check-expect (run-over-with-car LIVING-DILLO)
  (make-dillo 3.8 #false))
(check-expect (run-over-with-car DEAD-DILLO)
  DEAD-DILLO)

(define (run-over-with-car d)
  (make-dillo (dillo-weight d) #false))
Programming with armadillos

Pick a representation for armadillos ("dillo" for short), where a dillo has a weight and may or may not be alive.

Implement `run-over-with-car`, which takes a dillo and returns a dead dillo of equal weight.

Implement `feed-dillo`, where a dillo eats two pounds of food at a time.
Programming with armadillos

Pick a representation for armadillos (“dillo” for short), where a dillo has a weight and may or may not be alive.

Implement **run-over-with-car**, which takes a dillo and returns a dead dillo of equal weight.

Implement **feed-dillo**, where a dillo eats two pounds of food at a time.

*Except*: Dead armadillos eat no food.
;; feed-dillo : Dillo -> Dillo
;; Feed an armadillo two lbs of food, if it’s alive.

(check-expect (feed-dillo LIVING-DILLO)
  (make-dillo 5.8 #true))
(check-expect (feed-dillo DEAD-DILLO)
  DEAD-DILLO)
;;; feed-dillo : Dillo -> Dillo
;;; Feed an armadillo two lbs of food, if it’s alive.

(check-expect (feed-dillo LIVING-DILLO)
  (make-dillo 5.8 #true))

(check-expect (feed-dillo DEAD-DILLO)
  DEAD-DILLO)

(define (feed-dillo d)
  (... (dillo-weight d) (dillo-alive? d)))
;; feed-dillo : Dillo -> Dillo
;; Feed an armadillo two lbs of food, if it’s alive.

(check-expect (feed-dillo LIVING-DILLO)
  (make-dillo 5.8 #true))

(check-expect (feed-dillo DEAD-DILLO)
  DEAD-DILLO)

(define (feed-dillo d)
  (if (dillo-alive? d)
      (make-dillo (+ 2 (dillo-weight d))
                   #true)
      d))
Expanding our zoo: Ants
An ant has

- a weight

- a location in the zoo

(0, 0)

(x, y)
(define-struct ant (weight loc))
;; An Ant is
;; (make-ant PositiveNumber Posn)
(define-struct ant (weight loc))
;; An Ant is
;;  (make-ant PositiveNumber Posn)
(define-struct ant (weight loc))
;; An Ant is
;;  (make-ant PositiveNumber Posn)

(make-ant 0.001 (make-posn 4 5))
(make-ant 0.007 (make-posn 3 17))
Ants

(make-ant 0.001 (make-posn 4 5))

(make-ant 0.007 (make-posn 3 17))
Programming with ants

Define `ant-at-home?`, which takes an ant and reports whether it’s at the origin.
Signature and purpose
Signature and purpose

;; ant-at-home? : Ant -> Boolean
Signature and purpose

;; ant-at-home? : Ant -> Boolean
;;   Check whether the given ant is home, i.e., at (0, 0)
Examples

;; ant-at-home? : Ant -> Boolean
;;  Check whether the given ant is home, i.e., at (0, 0)

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
;; ant-at-home? : Ant -> Boolean
;; Check whether the given ant is home, i.e., at (0, 0)
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (ant-loc a) ...))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
Template

;; ant-at-home? : Ant -> Boolean
;; Check whether the given ant is home, i.e., at (0, 0)
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

New template rule:
Data definition reference \(\Rightarrow\) template reference

Add templates for referenced data, if needed, and implement body for referenced data.

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
template

;; ant-at-home? : Ant -> Boolean
;; Check whether the given ant is home, i.e., at (0, 0)
(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

(define (posn-at-home? p)
  (... (posn-x p) ... (posn-y p) ...))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
;;; ant-at-home? : Ant -> Boolean
;;; Check whether the given ant is home, i.e., at (0, 0)
(define (ant-at-home? a)
  (posn-at-home? (ant-loc a)))

(define (posn-at-home? p)
  (and (= (posn-x p) 0)
       (= (posn-y p) 0)))

(check-expect (ant-at-home? (make-ant 0.001 (make-posn 0 0))) #true)
(check-expect (ant-at-home? (make-ant 0.001 (make-posn 1 1))) #false)
Shapes of data and templates

The shape of the template matches the shape of the data

;;; An Ant is
;;;   (make-ant Number Posn)

;;; A Posn is
;;;   (make-posn Number Number)

(define (ant-at-home? a)
  (... (ant-weight a)
       ... (posn-at-home? (ant-loc a)) ...))

(define (posn-at-home? p)
  (... (posn-x p) ... (posn-y p) ...))
Programming with ants

Define **feed-ant**, which feeds ant 0.001 lbs of food.

Define **move-ant**, which takes an ant, an amount to move X, and an amount to move Y, and returns a moved ant.
Abstracting the zoo: Animals
Animals

All animals need to eat…

Define *feed-animal*, which takes an animal (a rabbit, a dillo, or an ant) and feeds it (0.25 lbs, 2 lbs, or 0.001 lbs, respectively).
Animals

All animals need to eat...

Define **feed-animal**, which takes an animal (a rabbit, a dillo, or an ant) and feeds it (0.25 lbs, 2 lbs, or 0.001 lbs, respectively).

What's an animal?
Animal data definition

;;; An Animal is either
;;; - Rabbit
;;; - Dillo
;;; - Ant

The “either” makes this a new kind of data definition: data with varieties or union data.

Examples:

(make-rabbit "Hopper" 2 "bananas")
(make-dillo 2 #true)
(make-ant 0.002 (make-posn 3 4))
Feeding animals

;; feed-animal : Animal -> Animal
;; Feed the given animal

(check-expect
 (feed-animal (make-rabbit "Hopper" 2 "bananas"))
 (make-rabbit "Hopper" 2.25 "bananas"))

(check-expect
 (feed-animal (make-dillo 2 #true))
 (make-dillo 4 #true))

(check-expect
 (feed-animal (make-ant 0.002 (make-posn 3 4)))
 (make-ant 0.003 (make-posn 3 4)))
Template for Animals

For the template step...

\[
(\text{define} \ (\text{feed-animal} \ a) \\
\ldots)
\]

Is a compound data?
Template for Animals

For the template step...

\[
(\text{define} \ (\text{feed-animal} \ a) \\
...)
\]

Is a compound data?

Technically yes, but the definition of \textit{Animal} doesn’t have \textit{make-something} so we don’t use the compound-data template rule.
Template for varieties

Three choices in the data definition

;;;; An Animal is either
;;;; - Rabbit
;;;; - Dillo
;;;; - Ant

means a cond in the template with three cases:

(define (feed-animal a)
  (cond [(rabbit? a) ...]
        [(dillo? a) ...]
        [(ant? a) ...]))
Template for varieties

Three choices in the data definition

;; An Animal is either
;; – Rabbit
;; – Dillo
;; – Ant

means a **cond** in the template with three cases:

```
(define (feed-animal a)
  (cond [(rabbit? a) ...]
        [(dillo? a) ...]
        [(ant? a) ...])
```

**New template rule:**
Varieties ⇒ cond
;;; feed-animal : Animal -> Animal
;;; Feed the given animal

(check-expect
 (feed-animal (make-rabbit "Hopper" 2 "bananas"))
 (make-rabbit "Hopper" 2.25 "bananas"))

(check-expect
 (feed-animal (make-dillo 2 #true))
 (make-dillo 4 #true))

(check-expect
 (feed-animal (make-ant 0.002 (make-posn 3 4)))
 (make-ant 0.003 (make-posn 3 4)))

(define (feed-animal a)
  (cond [(rabbit? a) (feed-rabbit a)]
        [(dillo? a) (feed-dillo a)]
        [(ant? a) (feed-ant a)]))
Acknowledgments

This lecture incorporates material from:


Marc Smith, Vassar College