Other Kinds of Data

Suppose we want to represent snakes:

• name
• weight
• favorite food

What kind of data is appropriate?
Other Kinds of Data

Suppose we want to represent snakes:

- name
- weight
- favorite food

What kind of data is appropriate?

Not `num`, `bool`, `string`, `image`, or `posn`...
Data Definitions and define-struct

Here’s what we’d like:

A *snake* is

`(make-snake string num string)`
Data Definitions and define-struct

Here’s what we’d like:

A **snake** is

\[ \text{make-snake string num string} \]

... but **make-snake** is not built into DrRacket
Data Definitions and define-struct

Here’s what we’d like:

A **snake** is

(make-snake string num string)

... but **make-snake** is not built into DrRacket

We can tell DrRacket about **snake**:

(define-struct snake (name weight food))
Data Definitions and define-struct

Here’s what we’d like:

A **snake** is

\[(\text{make-snake} \text{ string num string})\]

... but **make-snake** is not built into DrRacket

We can tell DrRacket about **snake**:

\[(\text{define-struct} \text{ snake} \text{ (name weight food)})\]

Creates the following:

- **make-snake**
- **snake-name**
- **snake-weight**
- **snake-food**
Data Definitions and define-struct

Here’s what we’d like:

A *snake* is

\[(\text{make-snake string num string})\]

... but *make-snake* is not built into DrRacket

We can tell DrRacket about *snake*:

\[(\text{define-struct snake (name weight food)})\]

Creates the following:

\[(\text{snake-name (make-snake X Y Z)}) \rightarrow X\]
\[(\text{snake-weight (make-snake X Y Z)}) \rightarrow Y\]
\[(\text{snake-food (make-snake X Y Z)}) \rightarrow Z\]
(define-struct snake (name weight food))

(snake "Slinky" 10 "rats"
  (make-snake "Slinky" 10 "rats"))

(snake "Slimey" 8 "pudding"
  (make-snake "Slimey" 8 "pudding"))

(define-struct posn (x y))

(posn 3 4
  (make-posn 3 4))

(posn 8 -2
  (make-posn 8 -2))
Deciding to define *snake* is in the first step of the design recipe
Data

Deciding to define `snake` is in the first step of the design recipe

**Handin artifact:** a comment and/or `define-struct`

```
; A snake is
; (make-snake string num string)

(define-struct snake (name weight food))
```
Data

Deciding to define **snake** is in the first step of the design recipe

**Handin artifact:** a comment and/or **define-struct**

; A snake is
; (make-snake string num string)

(define-struct snake (name weight food))

Now that we’ve defined **snake**, we can use it in signatures
Programming with Snakes

Implement \texttt{snake-skinny?}, which takes a snake and returns \texttt{true} if the snake weighs less than 10 pounds, \texttt{false} otherwise
Implement `snake-skinny?`, which takes a snake and returns `true` if the snake weights less than 10 pounds, `false` otherwise.

Implement `feed-snake`, which takes a snake and returns a snake with the same name and favorite food, but five pounds heavier.
Programming with Armadillos

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

Implement \texttt{feed-dillo}, where a dillo eats 2 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 2 pounds of food at a time

... unless it's dead
Expanding the Zoo

We have snakes and armadillos. Let’s add ants.

An ant has

• a weight

• a location in the zoo
Expanding the Zoo

We have snakes and armadillos. Let’s add ants.

An ant has

• a weight

• a location in the zoo

; An ant is
; (make-ant num posn)
(define-struct ant (weight loc))
Expanding the Zoo

We have snakes and armadillos. Let’s add ants.

An ant has

• a weight

• a location in the zoo

; An ant is
; (make-ant num posn)
(define-struct ant (weight loc))

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

\[
\text{(make-ant 0.001 (make-posn 4 5))}
\]

\[
\text{(make-ant 0.007 (make-posn 3 17))}
\]
Programming with Ants

Define \texttt{ant-at-home?}, which takes an ant and reports whether it is at the origin.
; ant -> bool
Signature, Purpose, and Header

; ant -> bool
; Check whether ant a is home
Signature, Purpose, and Header

; ant -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...)

(ant-at-home? a)
Examples

; ant -> bool
; Check whether ant a is home
(define (ant-at-home? 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 -> bool
; Check whether ant a is home
(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 -> bool
; Check whether ant a is home
(define (ant-at-home? a)
  ...
  (ant-weight a)
  ...
  (posn-at-home? (ant-loc a)) ...)

New template rule: data-defn reference ⇒ 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 -> bool
; Check whether ant a is home
(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)
(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) ...)

(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 num posn)

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

(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) ...)

Define **feed-ant**, which feeds an 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
Animals

All animals need to eat...

Define `feed-animal`, which takes an animal (snake, dillo, or ant) and feeds it (5 lbs, 2 lbs, or 0.001 lbs, respectively)
All animals need to eat...

Define *feed-animal*, which takes an animal (snake, dillo, or ant) and feeds it (5 lbs, 2 lbs, or 0.001 lbs, respectively)

What is an *animal*?
Animal Data Definition

; An animal is either
;   - snake
;   - dillo
;   - ant
Animal Data Definition

; An animal is either
; - snake
; - dillo
; - ant

The “either” above makes this a new kind of data definition:

data with varieties
Animal Data Definition

; An animal is either
;    - snake
;    - dillo
;    - ant

The “either” above makes this a new kind of data definition:

data with varieties

Examples:

(make-snake "slinky" 10 "rats")

(make-dillo 2 #true)

(make-ant 0.002 (make-posn 3 4))
Feeding Animals

; animal -> animal
; To feed the animal a
(define (feed-animal a)
  ...)

Feeding Animals

; animal -> animal
; To feed the animal a
(define (feed-animal a) ...

(check-expect (feed-animal (make-snake "Slinky" 10 "rats")))
(make-snake "Slinky" 15 "rats"))

(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 \textit{a} compound data?
Template for Animals

For the template step...

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

- Is \textit{a} compound data?

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

Choice in the data definition

; An animal is either
; - snake
; - dillo
; - ant

means cond in the template:

(define (feed-animal a)
  (cond
    [...  ... ]
    [...  ... ]
    [...  ... ]
    [...  ... ]))

Three data choices means three cond cases
Questions for Varieties

(define (feed-animal a)
  (cond
    [.... [...]
    [.... [...]
    [.... [...]])

How do we write a question for each case?
Questions for Varieties

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

How do we write a question for each case?

It turns out that

\[
\text{(define-struct \textbf{snake} \textbf{name weight food})}
\]
provides \textbf{snake}? 

\[
\text{(snake? \textbf{(make-snake} "\textbf{slinky}" 5 "\textbf{rats}"))} \rightarrow \textbf{#true}
\]
\[
\text{(snake? \textbf{(make-dillo} 2 \textbf{#true})}) \rightarrow \textbf{#false}
\]
\[
\text{(snake? 17)} \rightarrow \textbf{#false}
\]
Template

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

New template rule: varieties ⇒ cond
(define (feed-animal a)
  (cond
    [(snake? a) ...]
    [(dillo? a) ...]
    [(ant? a) ...]))

New template rule: varieties ⇒ cond

Now continue template case-by-case...
Template

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

Remember: references in the data definition ⇒ template references
(define (feed-animal a)
  (cond
    [(snake? a) ... (feed-snake a) ...]
    [(dillo? a) ... (feed-dillo a) ...]
    [(ant? a) ... (feed-ant a) ...]))

Remember: references in the data definition ⇒ template references

; An animal is either
; - snake
; - dillo
; - ant
Shapes of Data and Templates

; An animal is either
;  - snake
;  - dillo
;  - ant

; A snake is
; (make-snake string num string)

; A dillo is
; (make-dillo num bool)

; An ant is
; (make-ant num posn)

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

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

(define (feed-snake s)
  ... (snake-name s) ... (snake-weight s)
  ... (snake-food s) ...)

(define (feed-dillo d)
  ... (dillo-weight d)
  ... (dillo-alive? d) ...)

(define (feed-ant a)
  ... (ant-weight d)
  ... (feed-posn (ant-loc d)) ...)

(define (feed-posn p)
  ... (posn-x p) ... (posn-y p) ...)

Design Recipe III

**Data**

- Understand the input data

**Signature, Purpose, and Header**

- Describe (but don’t write) the function

**Examples**

- Show what will happen when the function is done

**Template**

- Set up the body based on the input data (and only the input)

**Body**

- The most creative step: implement the function body

**Test**

- Run the examples
Data

When the problem statement mentions \( N \) different varieties of a thing, write a data definition of the form

```
; A thing is
;    - variety1
;    ...
;    - varietyN
```
# Design Recipe III

## Data
- Understand the input data

## Signature, Purpose, and Header
- Describe (but don’t write) the function

## Examples
- Show what will happen when the function is done

## Template
- Set up the body based on the input data (and *only* the input)

## Body
- The most creative step: implement the function body

## Test
- Run the examples
Examples

When the input data has varieties, be sure to pick each variety at least once.

; An animal is either
; - snake
; - dillo
; - ant

(check-expect (feed-animal (make-snake "Slinky" 10 "rats"))
(make-snake "Slinky" 15 "rats"))

(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)))
## Design Recipe III

### Data
- Understand the input data

### Signature, Purpose, and Header
- Describe (but don’t write) the function

### Examples
- Show what will happen when the function is done

### Template
- Set up the body based on the input data (and only the input)

### Body
- The most creative step: implement the function body

### Test
- Run the examples
When the input data has varieties, start with cond

- \( N \) varieties \( \Rightarrow \) \( N \) cond lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

\[
\text{(define (feed-animal a)}
\text{(cond}
\text{[(snake? a) ...]}
\text{[(dillo? a) ...]}
\text{[(ant? a) ...])})
\]
When the input data has varieties, start with `cond`

- **N** varieties ⇒ **N** `cond` lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

When the data definition refers to a data definition, make the template refer to a template

```
(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) ...
)
```
Template

When the input data has varieties, start with \texttt{cond}

- \texttt{N} varieties $\Rightarrow$ \texttt{N cond} lines
- Formulate a question to match each corresponding variety
- Continue template steps case-by-case

When the data definition refers to a data definition, make the template refer to a template

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