Compound Data So Far

A posn is

\[(\text{make-posn } X \ Y)\]

where \(X\) is a \text{num} and \(Y\) is a \text{num}

- \((\text{make-posn } 1 \ 2)\) is a value
- \((\text{posn-x } (\text{make-posn } 1 \ 2)) \rightarrow 1\)
- \((\text{posn-y } (\text{make-posn } 1 \ 2)) \rightarrow 2\)
Compound Data So Far

A posn is

\[(\text{make-posn } X \ Y)\]

where \(X\) is a num and \(Y\) is a num

- \((\text{make-posn } 1 \ 2)\) is a value
- \((\text{posn-x } (\text{make-posn } 1 \ 2)) \rightarrow 1\)
- \((\text{posn-y } (\text{make-posn } 1 \ 2)) \rightarrow 2\)

So much for computation... how about program design?
Design Recipe I

Data
• Understand the input data: `num`, `bool`, `string`, or `image`

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

Examples
• Show what will happen when the function is done

Body
• The most creative step: implement the function body

Test
• Run the examples
If the input is compound data, start the body by selecting the parts
If the input is compound data, start the body by selecting the parts

; posn -> num
; Return the X part of p if it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  ...)

(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
If the input is compound data, start the body by selecting the parts

; posn -> num
; Return the X part of p if it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
    ... (posn-x p) ... (posn-y p) ...)

(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
If the input is compound data, start the body by selecting the parts

; posn -> num
; Return the X part of p if it's bigger
; than the Y part, otherwise the Y part
(define (max-part p)
  (cond
   [ (> (posn-x p) (posn-y p)) (posn-x p)]
   [else (posn-y p)])
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
If the input is compound data, start the body by selecting the parts

; posn -> num
; Return the X part of p if it's bigger than the Y part, otherwise the Y part
(define (max-part p)
    (cond
     [(> (posn-x p) (posn-y p)) (posn-x p)]
     [else (posn-y p)])
)
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)

Since this guideline applies before the usual body work, let’s split it into an explicit step
Design Recipe II

**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
If the input is compound data, start the body by selecting the parts

; posn -> num
; ...
(define (max-part p)
  ... (posn-x p) ... (posn-y p) ...)

10
If the input is compound data, start the body by selecting the parts

\[
\text{; posn} \rightarrow \text{num} \\
\text{; ...} \\
(\text{define } (\text{max-part } p)) \\
\text{ ... (posn-x } p) \text{ ... (posn-y } p) \text{ ...)}
\]

**Check:** number of parts in template = number of parts data definition named in contract
If the input is compound data, start the body by selecting the parts

; posn -> num
; ...
(define (max-part p)
 ... (posn-x p) ... (posn-y p) ...)

Check: number of parts in template =
number of parts data definition named in contract

A posn is

(make-posn X Y)

where X is a num and Y is a num
If the input is compound data, start the body by selecting the parts

Handin artifact: a comment

; posn -> num
; Return the X part of p if it's bigger than the Y part, otherwise the Y part
;
; (define (max-part p)
; ... (posn-x p) ... (posn-y p) ...)
(define (max-part p)
  ... (posn-x p) ... (posn-y p) ...)
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
Design Recipe II

**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
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

(\texttt{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

$$(\text{make-snake }\text{string }\text{num }\text{string})$$

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

We can tell DrRacket about **snake**:

$$(\text{define-struct snake (name weight food)})$$
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:

- **make-snake**
- **snake-name**
- **snake-weight**
- **snake-food**
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))

Creates the following:

≡ (snake-name (make-snake X Y Z)) → X
≡ (snake-weight (make-snake X Y Z)) → Y
≡ (snake-food (make-snake X Y Z)) → Z
(define-struct snake (name weight food))

(define snake 1)

(define snake 2)

(define (make-snake name weight food)
  ...)

(define (make-posn x y)
  ...)

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

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

(make-posn 3 4)

(make-posn 8 -2)
Data

Deciding to define *snake* is in the first step of the design recipe
Deciding to define \texttt{snake} is in the first step of the design recipe

Handin artifact: a comment and/or \texttt{define-struct}

\begin{verbatim}
; A snake is
; (make-snake string num string)

(define-struct snake (name weight food))
\end{verbatim}
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 contracts
Programming with Snakes

Implement `snake-skinny?`, which takes a snake and returns `true` if the snake weights less than 10 pounds, `false` otherwise.
Programming with Snakes

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
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.
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
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