- **posn**
  - **define-struct**

Compound Data So Far

A **posn** is

```
(make-posn num num)
```

- `(make-posn 1 2)` is a value
- `(posn-x (make-posn 1 2))` → `1`
- `(posn-y (make-posn 1 2))` → `2`

So much for computation... how about program design?

Body

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

Body

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

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


(max-part (make-posn 10 11)) "should be" 11
(max-part (make-posn 7 5)) "should be" 7
```

Body

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

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


(max-part (make-posn 10 11)) "should be" 11
(max-part (make-posn 7 5)) "should be" 7
```

Body

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

```
; max-part : posn -> num
; Return the X part of p is 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)]))
(max-part (make-posn 10 11)) "should be" 11
(max-part (make-posn 7 5)) "should be" 7
```

Body

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

```
; max-part : posn -> num
; Return the X part of p is 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)]))
(max-part (make-posn 10 11)) "should be" 11
(max-part (make-posn 7 5)) "should be" 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

Contract, 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

~~Body~~ Template

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

```
; max-part : 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 num num)
```

~~Body~~ Template

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

Handin artifact: a comment (required starting with HW 3)

```
; max-part : posn -> num
; Return the X part of p is 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) ...)
(max-part (make-posn 10 11)) "should be" 11
(max-part (make-posn 7 5)) "should be" 7
```

- **posn**
- **define-struct**

Other Kinds of Data

Suppose we want to represent snakes:

- name
- weight
- favorite food

What kind of data is appropriate?

Not **`num`**, **`bool`**, **`sym`**, **`image`**, or **`posn`**...

Data Definitions and define-struct

Here's what we'd like:

A **snake** is
(make-snake sym num sym)

But **make-snake** is not built into DrScheme

We can tell DrScheme about **snake**:

```
(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 sym num sym)

But **make-snake** is not built into DrScheme

We can tell DrScheme 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
```

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 sym num sym)

(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
- 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
- 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