# 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)
  ...)


(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 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) ...)


(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 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)]))
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 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)]))
(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

**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

In definitions: a comment

```
; 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) ...)
(check-expect (max-part (make-posn 10 11)) 11)
(check-expect (max-part (make-posn 7 5)) 7)
```