How to Design Programs

http://www.htdp.org
How to Design Programs

• Determine the **representation**
  ○ data definition or **define-type**

• Write **examples**
  ○ **test**

• Create a **template** for the implementation
  ○ **cond** or **type-case**, if variants
  ○ extract field values, if any
  ○ cross- and self-calls, if data references

• Finish **body** implementation case-by-case

• Run **tests**
Representation

• Keep track of the number of cookies in a cookie jar

  \texttt{number}

  \texttt{eat-cookie : (number \rightarrow number)}
Examples

number

eat-cookie : (number -> number)

(test (eat-cookie 10) 9)
(test (eat-cookie 1) 0)
(test (eat-cookie 0) 0)
Template

number

eat-cookie : (number -> number)

(define (eat-cookie [n : number])
  ... n ...)

Body

number

eat-cookie : (number -> number)

(define (eat-cookie [n : number])
  ... n ...)

(test (eat-cookie 10) 9)
(test (eat-cookie 1) 0)
(test (eat-cookie 0) 0)
Body

number

eat-cookie : (number -> number)

(define (eat-cookie [n : number])
  (if (> n 0)
      (- n 1)
      0))

(test (eat-cookie 10) 9)
(test (eat-cookie 1) 0)
(test (eat-cookie 0) 0)
Representation

• Track a position on the screen

```
(define-type Posn
  [posn (x : number)
    (y : number)])

flip : (Posn -> Posn)
```
Examples

(define-type Posn
  [
    posn (x : number)
    (y : number)
  ])

flip : (Posn -> Posn)

(test (flip (posn 1 17)) (posn 17 1))
(test (flip (posn -3 4)) (posn 4 -3))
Template

(define-type Posn
  [posn (x : number)
    (y : number)])

flip : (Posn -> Posn)

(define (flip [p : Posn])
  ... (posn-x p)
  ... (posn-y p) ...

  or

(define (flip [p : Posn])
  (type-case Posn p
    [posn (x y) ... x ... y ...]))
(define-type Posn
  [posn (x : number)
    (y : number)]
)

flip : (Posn -> Posn)

(define (flip [p : Posn])
  (type-case Posn p
    [posn (x y) ... x ... y ...]))

(test (flip (posn 1 17)) (posn 17 1))
(test (flip (posn -3 4)) (posn 4 -3))
Body

(define-type Posn
  [posn (x : number)
    (y : number)])

flip : (Posn -> Posn)

(define (flip [p : Posn])
  (type-case Posn p
    [posn (x y) (posn y x)]))

(test (flip (posn 1 17)) (posn 17 1))
(test (flip (posn -3 4)) (posn 4 -3))
Representation

• Track an ant, which has a location and a weight

\[
\text{(define-type Ant}
\begin{array}{l}
\text{ant (location : Posn)} \\
\text{(weight : number)}
\end{array}
\text{)}
\]

\[
\text{ant-at-home? : (Ant -> boolean)}
\]
Examples

(define-type Ant
  [ant (location : Posn)
     (weight : number)])

ant-at-home? : (Ant -> boolean)

(test
  (ant-at-home? (ant (posn 0 0) 0.0001))
  #t)
(test
  (ant-at-home? (ant (posn 5 10) 0.0001))
  #f)
Template

(define-type Ant
  [ant (location : Posn)
      (weight : number)]
)

ant-at-home? : (Ant -> boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [ant (loc wgt)
        ... loc ...
        ... wgt ...]))
Template

(define-type Ant
    [ant (location : Posn)
        (weight : number)])

ant-at-home? : (Ant -> boolean)

(define (ant-at-home? [a : Ant])
    (type-case Ant a
        [ant (loc wgt)
            ... (is-home? loc) ...
            ... wgt ...]))

(define (is-home? [p : Posn])
    (type-case Posn p
        [posn (x y) ... x ... y ...]))
(define-type Ant
    [ant (location : Posn)
        (weight : number)])

ant-at-home? : (Ant -> boolean)

(define (ant-at-home? [a : Ant])
    (type-case Ant a
        [ant (loc wgt)
            ... (is-home? loc) ...
            ... wgt ...]))

(define (is-home? [p : Posn])
    (type-case Posn p
        [posn (x y) ... x ... y ...])))
(define-type Ant
  [ant (location : Posn)
   (weight : number)])

ant-at-home? : (Ant -> boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [ant (loc wgt) (is-home? loc)]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [posn (x y) ... x ... y ...]))
Body

(define-type Ant
  [ant (location : Posn)
    (weight : number)])

ant-at-home? : (Ant -> boolean)

(define (ant-at-home? [a : Ant])
  (type-case Ant a
    [ant (loc wgt) (is-home? loc)]))

(define (is-home? [p : Posn])
  (type-case Posn p
    [posn (x y) (and (zero? x)
                    (zero? y)])])
• Track an animal, which is a snake or a tiger

```
(define-type Animal
  [snake (name : symbol)
    (weight : number)
    (food : symbol)]
  [tiger (name : symbol)
    (stripe-count : number)])

heavy-animal? : (Animal -> boolean)
```
Examples

(define-type Animal
  [snake (name : symbol)
    (weight : number)
    (food : symbol)]
  [tiger (name : symbol)
    (stripe-count : number)])

heavy-animal? : (Animal -> boolean)

(test (heavy-animal? (snake 'Slinky 10 'rats))
  #t)
(test (heavy-animal? (snake 'Slimey 8 'cake))
  #f)
(test (heavy-animal? (tiger 'Tony 14))
  #t)
Template

(define-type Animal
  [snake (name : symbol)
    (weight : number)
    (food : symbol)]
  [tiger (name : symbol)
    (stripe-count : number)]
)

heavy-animal? : (Animal -> boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [snake (n w f)
      ... n ... w ...
      ... f ...]
    [tiger (n sc)
      ... n ... sc ...])))
Body

(define-type Animal
  [snake (name : symbol)
    (weight : number)
    (food : symbol)]
  [tiger (name : symbol)
    (stripe-count : number)])

heavy-animal? : (Animal -> boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [snake (n w f)
        ... n ... w ...
        ... f ...]
    [tiger (n sc)
        ... n ... sc ...]))
(define-type Animal
    [snake (name : symbol)
        (weight : number)
        (food : symbol)]
    [tiger (name : symbol)
        (stripe-count : number)])

heavy-animal? : (Animal -> boolean)

(define (heavy-animal? [a : Animal])
    (type-case Animal a
        [snake (n w f) (>= w 10)]
        [tiger (n sc)
            ...
            n ...
            sc ...
            ]))
Body

(define-type Animal
  [snake (name : symbol)
    (weight : number)
    (food : symbol)]
  [tiger (name : symbol)
    (stripe-count : number)])

heavy-animal? : (Animal -> boolean)

(define (heavy-animal? [a : Animal])
  (type-case Animal a
    [snake (n w f) (> w 10)]
    [tiger (n sc) t])))
Representation

- Track an aquarium, which has any number of fish, each with a weight

```
(define-type ListOfNumber
  [emptyList]
  [biggerList (n : number)
    (rest : ListOfNumber)])

feed-fish : (ListOfNumber -> ListOfNumber)
```
Representation

• Track an aquarium, which has any number of fish, each with a weight

; A (listof 'a) is either
; - empty
; - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))
Examples

; A (listof 'a) is either
;  - empty
;  - (cons v r)
;  where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(test (feed-fish (list)) (list))
(test (feed-fish (list 1 2 3)) (list 2 3 4))
Template

; A (listof 'a) is either
; - empty
; - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
  (cond
    [(empty? lon) ...]
    [(cons? lon) ...]))
Template

; A (listof 'a) is either
;  - empty
;  - (cons v r)
;  where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

 DEFINE (feed-fish [lon : (listof number)])
   (cond
    [(empty? lon) ...]
    [(cons? lon)
     ... (first lon) ...
     ... (rest lon) ...]])
Template

; A (listof 'a) is either
; - empty
; - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
   (cond
    [(empty? lon) ...]
    [(cons? lon)
      ... (first lon) ... 
      ... (feed-fish (rest lon)) ...])
Body

; A (listof 'a) is either
;   - empty
;   - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
  (cond
    [(empty? lon) ...]
    [(cons? lon)
      ... (first lon) ...
      ... (feed-fish (rest lon)) ...])

```
Body

; A (listof 'a) is either
;   - empty
;   - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
(cond
  [(empty? lon) empty]
  [(cons? lon)
   ... (first lon) ...
   ... (feed-fish (rest lon)) ...])]
; A (listof 'a) is either
;   - empty
;   - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
(define (feed-fish [lon : (listof number)]))
  (cond
   [(empty? lon) empty]
   [(cons? lon)
    ... (+ 1 (first lon)) ...
    ... (feed-fish (rest lon)) ...)])
Body

; A (listof 'a) is either
; - empty
; - (cons v r)
; where v : 'a and r : (listof 'a)

feed-fish : ((listof number) -> (listof number))

(define (feed-fish [lon : (listof number)]))
  (cond
    [(empty? lon) empty]
    [(cons? lon)
     (cons (+ 1 (first lon))
          (feed-fish (rest lon)))]))
Implementation Matches Data

; A (listof number) is either
;  - empty
;  - (cons number (listof number))

(define (feed-fish [lon : (listof number)]))
  (cond
    [(empty? lon) ...]
    [(cons? lon) ... (first lon)
     ... (feed-fish (rest lon)) ...]])
How to Design Programs
More Examples
GUls

Possible programs:

- Can click?
- Find a label
- Read screen
Representing GUls

- labels
  - a label string
- buttons
  - a label string
  - enabled state
- lists
  - a list of choice strings
  - selected item

(define-type GUI
  [label (text : string)]
  [button (text : string)
    (enabled? : boolean)]
  [choice (items : (listof string))
    (selected : number)])
Read Screen

- Implement `read-screen`, which takes a GUI and returns a list of strings for all the GUI element labels
(define (read-screen [g : GUI]) : (listof string)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]))

(test (read-screen (label "Hi")))
  (list "Hi")
(test (read-screen (button "Ok" true)))
  (list "Ok")
(test (read-screen (choice (list "Apple" "Banana")
                          0)))
  (list "Apple" "Banana")
Assemblies GUIs

- label
- buttons
- lists
- vertical stacking
  - two sub-GUIs
- horizontal stacking
  - two sub-GUIs

(define-type GUI
  [label (text : string)]
  [button (text : string)
    (enabled? : boolean)]
  [choice (items : (listof string))
    (selected : number)]
  [vertical (top : GUI)
    (bottom : GUI)]
  [horizontal (left : GUI)
    (right : GUI)]
Assemblings GUls

• label
• buttons
• lists
• vertical stacking
  ○ two sub-GUls
• horizontal stacking
  ○ two sub-GUls

(define guil
  (vertical
    (horizontal
      (label "Pick a fruit:"))
    (choice
      (list "Apple" "Banana" "Coconut")
      0))
  (horizontal
    (button "Ok" false)
    (button "Cancel" true))))
Read Screen

- Implement `read-screen`, which takes a GUI and returns a list of strings for all the GUI element labels
Read Screen

(define (read-screen [g : GUI]) : (listof string)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]
    [vertical (t b) (append (read-screen t)
                             (read-screen b))]
    [horizontal (l r) (append (read-screen l)
                              (read-screen r))])))

...  
(test (read-screen guil)
  (list "Pick a fruit:"
       "Apple" "Banana" "Coconut"
       "Ok" "Cancel")))
(define-type GUI
  [label (text : string)]
  [button (text : string)
    (enabled? : boolean)]
  [choice (items : (listof string))
    (selected : number)]
  [vertical (top : GUI)
    (bottom : GUI)]
  [horizontal (left : GUI)
    (right : GUI)])

(define (read-screen [g : GUI]) : (listof string)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]
    [vertical (t b) (append (read-screen t)
                              (read-screen b))]
    [horizontal (l r) (append (read-screen l)
                               (read-screen r))])))
Design Steps

• Determine the representation
  ○ define-type, maybe

• Write examples
  ○ test

• Create a template for the implementation
  ○ type-case plus natural recursion, check shape!

• Finish body implementation case-by-case
  ○ usually the interesting part

• Run tests
Enable Button

- Implement **enable-button**, which takes a GUI and a string and enables the button whose name matches the string
Enable Button

The `name` argument is “along for the ride”:

```
(define (enable-button [g : GUI] [name : string]) : GUI
  (type-case GUI g
    [label (t) g]
    [button (t e?) (cond
      [(equal? t name) (button t true)]
      [else g])]
    [choice (i s) g]
    [vertical (t b) (vertical (enable-button t name)
      (enable-button b name))]
    [horizontal (l r) (horizontal (enable-button l name)
      (enable-button r name)))])
...
(test (enable-button guil "Ok")
  (vertical
    (horizontal (label "Pick a fruit:"))
    (choice (list "Apple" "Banana" "Coconut") 0))
  (horizontal (button "Ok" true)
    (button "Cancel" true))))
```
Show Depth

(test (show-depth
(I Hello
(2 Ok 2 Cancel
))
Hello
(Ok Cancel
))
)
Show Depth

Template:

```
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g

  [label (t) ... t ...]
  [button (t e?) ... t ... e? ...]
  [choice (i s) ... i ... s ...]
  [vertical (t b) ... (show-depth t)
    ... (show-depth b) ...]
  [horizontal (l r) ... (show-depth l)
    ... (show-depth r) ...]])

  (show-depth Ok)  ->  0 Ok
```
Show Depth

Template:

```scheme
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g
    [label (t) ... t ...]
    [button (t e?) ... t ... e? ...]
    [choice (i s) ... i ... s ...]
    [vertical (t b) ... (show-depth t)
         ... (show-depth b) ...]
    [horizontal (l r) ... (show-depth l)
       ... (show-depth r) ...]))

(show-depth Ok Cancel) → ... 0 Ok ... 0 Cancel ...
```

83
Show Depth

Template:

```
(define (show-depth [g : GUI]) : GUI
  (type-case GUI g
    [label (t) ... t ...]
    [button (t e?) ... t ... e? ...]
    [choice (i s) ... i ... s ...]
    [vertical (t b) ... (show-depth t)
      ... (show-depth b) ...]
    [horizontal (l r) ... (show-depth l)
      ... (show-depth r) ...]))
```

recursion results don’t have the right labels...
Show Depth

The \textit{n} argument is an \textit{accumulator}:

\begin{verbatim}
(define (show-depth-at [g : GUI] [n : number]) : GUI
  (type-case GUI g
    [label (t) (label (prefix n t))]
    [button (t e?) (button (prefix n t) e?)]
    [choice (i s) g]
    [vertical (t b) (vertical (show-depth-at t (+ n 1))
                        (show-depth-at b (+ n 1)))]
    [horizontal (l r) (horizontal (show-depth-at l (+ n 1))
                           (show-depth-at r (+ n 1)))]
  )

(define (show-depth [g : GUI]) : GUI
  (show-depth-at g 0))
\end{verbatim}
How to Design Programs

• Follow the design steps
• Use accumulators when necessary
• Reuse functions and/or “wish” for helpers