Part 1: Anonymous Functions

Anonymous Functions

value	definition
`3`	`(define pi 3)`
`'(1 2 3)`	`(define nums '(1 2 3))`
`(tiger 'Tony 14)`	`(define tony` `  (tiger 'Tony 14))`
`(lambda (x)` `  (+ x 1))`	`(define (f x)` `  (+ x 1))`

Applying Functions

```
((lambda (x) (+ x 1)) 10) → (+ 10 1)

(define (f x) (+ x 1))
(f 10)
→ (+ 10 1)

(define f (lambda (x) (+ x 1)))
(f 10)
→ ((lambda (x) (+ x 1)) 10)
→ (+ 10 1)
```

Using Anonymous Functions

```
(map add1 '(1 2 3)) → '(2 3 4)

(map (lambda (x) (+ x 3))
     '(1 2 3))                  → '(4 5 6)


(andmap even? '(2 4 6)) → #t

(andmap (lambda (x)
          (integer? (sqrt x)))
        '(4 9 16))                  → true
```

Part 2: 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

```
; number

; eat-cookie : number -> number
```

Representation

- Track a position on the screen

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

; flip : Posn -> Posn
```

Representation

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

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

; ant-at-home? : Ant -> boolean
```

Representation

- 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
```

Representation

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

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number
```

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**

Examples

```
; number

; eat-cookie : number -> number

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

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)))
```

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)
```

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)
```

Examples

```
; A list-of-number is either
;   - empty
;   - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

(test (feed-fish '()) '())
(test (feed-fish '(1 2 3)) '(2 3 4))
```

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**

Template

```
; number

; eat-cookie : number -> number

(define (eat-cookie n)
  ... n ...)
```

Template

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

; flip : Posn -> Posn

(define (flip p)
  ... (posn-x p)
  ... (posn-y p) ...)
```

or

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

Template

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

; ant-at-home? : Ant -> boolean

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

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

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)
  (type-case Animal a
    [snake (n w f)
           ... n ... w ...
           ... f ...]
    [tiger (n sc)
           ... n ... sc ...]))
```

Template

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

(define (feed-fish lon)
  (cond
   [(empty? lon) ...]
   [(cons? lon) ...]))
```

Template

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

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

Template

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

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

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**

Body

```
; number

; eat-cookie : number -> number

(define (eat-cookie n)
  ... n ...)
```

Body

```
; number

; eat-cookie : number -> number

(define (eat-cookie n)
  (if (> n 0)
      (- n 1)
      0))
```

Body

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

; flip : Posn -> Posn

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

Body

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

; flip : Posn -> Posn

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

Body

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

; ant-at-home? : Ant -> boolean

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

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

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

(define (is-home? p)
  (type-case Posn p
    [posn (x y) (and (zero? x)
                     (zero? y))]))
```

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)
  (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)
  (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)
  (type-case Animal a
    [snake (n w f) (>= w 10)]
    [tiger (n sc) #t]))
```

Body

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

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

Body

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

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

Body

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

(define (feed-fish lon)
  (cond
   [(empty? lon) empty]
   [(cons? lon)
    ... (+ 1 (first lon)) ...
    ... (feed-fish (rest lon)) ...]))
```

Body

```
; A list-of-number is either
;  - empty
;  - (cons number list-of-number)

; feed-fish : list-of-number
; -> list-of-number

(define (feed-fish lon)
  (cond
   [(empty? lon) empty]
   [(cons? lon)
    (cons (+ 1 (first lon))
          (feed-fish (rest lon)))]))
```

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**

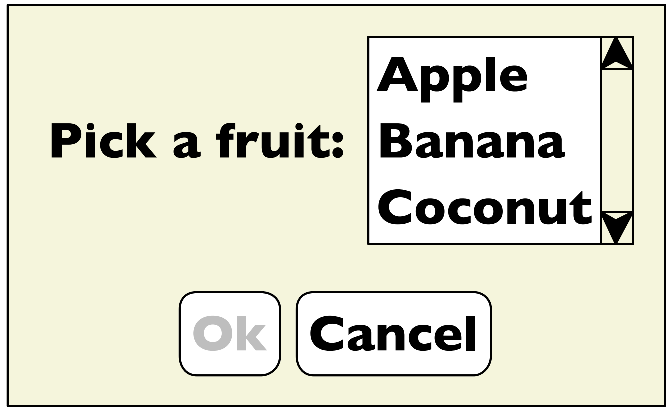
Implementation Matches Data

```
; A list-of-number is either
;   - empty
;   - (cons number list-of-number)

; feed-fish : list-of-number -> list-of-number
(define (feed-fish lon)
  (cond
   [(empty? lon) ...]
   [(cons? lon) ... (first lon)
    ... (feed-fish (rest lon)) ...]))
```

Part 3: GUI Examples

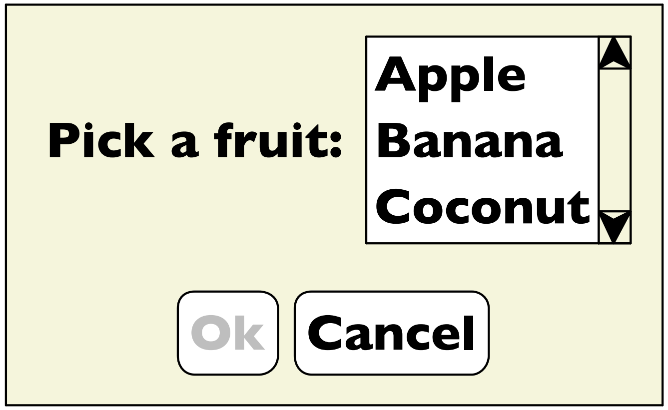
GUIs



Possible programs:

- Can click?
- Find a label
- Read screen

Representing GUIs



- 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 integer?)])
```

Read Screen

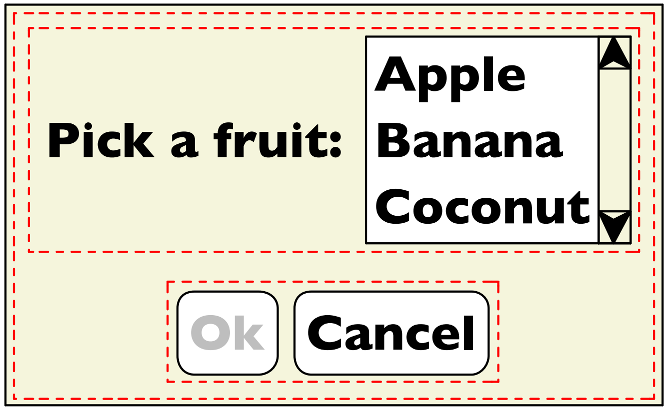
- Implement **`read-screen`**, which takes a GUI and returns a list of strings for all the GUI element labels

Read Screen

```
; read-screen : GUI -> list-of-string
(define (read-screen g)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]))

(test (read-screen (label "Hi"))
      '("Hi"))
(test (read-screen (button "Ok" true))
      '("Ok"))
(test (read-screen (choice '("Apple" "Banana") 0))
      '("Apple" "Banana"))
```

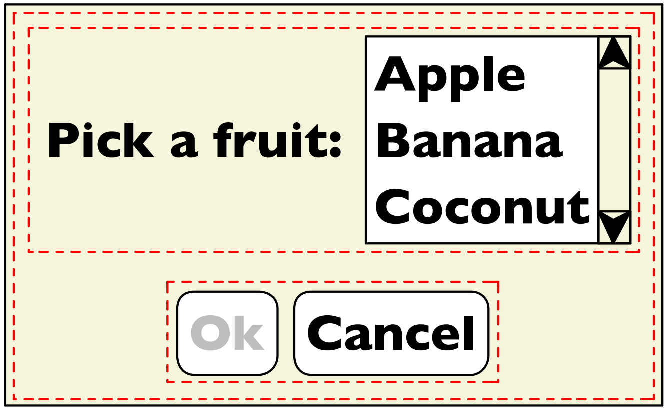
Assemblings GUIs



- 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 integer?)]
  [vertical (top GUI?)
            (bottom GUI?)]
  [horizontal (left GUI?)
              (right GUI?)])
```

Assemblings GUIs



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

```
(define gui1
  (vertical
   (horizontal
    (label "Pick a fruit:")
    (choice '("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

```
; read-screen : GUI -> list-of-string
(define (read-screen g)
  (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 gui1
      '("Pick a fruit:"
        "Apple" "Banana" "Coconut"
        "Ok" "Cancel"))
```

Function and Data Shapes Match

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

```
(define (read-screen g)
  (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

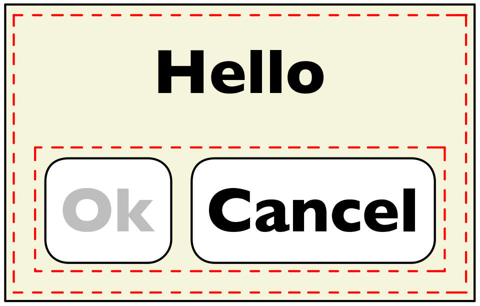
- Implement **`enable-button`**, which takes a GUI and a string and enables the button whose name matches the string

Enable Button

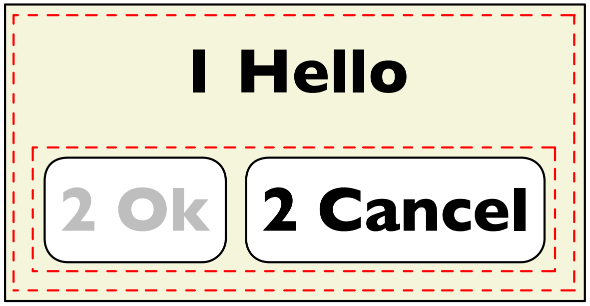
The **name** argument is "along for the ride":

```
; enable-button : GUI string -> GUI
(define (enable-button g name)
  (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 gui1 "Ok")
      (vertical
       (horizontal (label "Pick a fruit:")
                   (choice '("Apple" "Banana" "Coconut") 0))
       (horizontal (button "Ok" true)
                   (button "Cancel" true))))
```

Show Depth

(test (show-depth 

)



)

Show Depth

Template:

```
(define (show-depth g)
  (type-case GUI g
    [label (t) ...]
    [button (t e?) ...]
    [choice (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:

```
(define (show-depth g)
  (type-case GUI g
    [label (t) ...]
    [button (t e?) ...]
    [choice (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 ...

Show Depth

Template:

```
(define (show-depth g)
  (type-case GUI g
    [label (t) ...]
    [button (t e?) ...]
    [choice (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 `n` argument is an *accumulator*:

```
; show-depth-at : GUI num -> GUI
(define (show-depth-at g n)
  (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)))]))

; show-depth : GUI -> GUI
(define (show-depth g)
  (show-depth-at g 0))
```

How to Design Programs

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