How to Design Programs

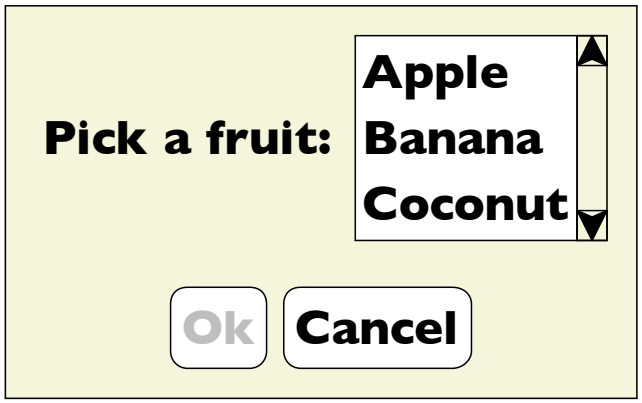
How to (in Racket):

- represent data
  - variants
  - trees and lists
- write functions that process the data

See also

`http://www.htdp.org/`

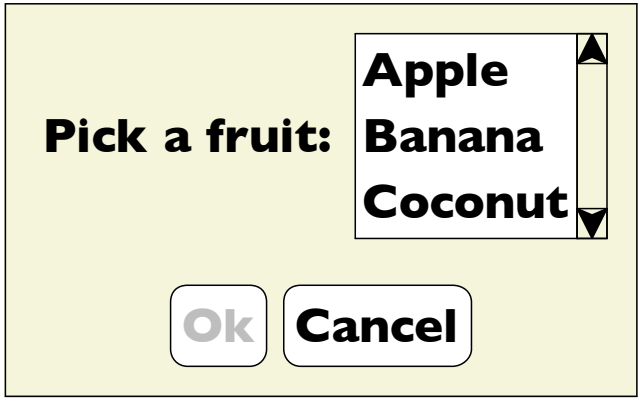
Running Example: GUIs



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

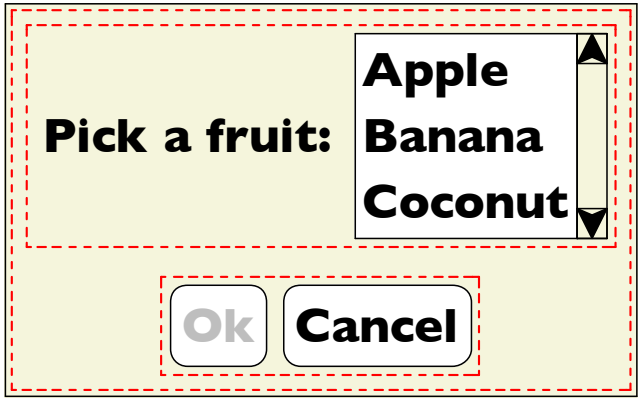
```
(define-type GUI
  [label (text string?)]
  [button (text string?)
          (enabled? boolean?)]
  [choice (items (listof string?))
          (selected integer?)])
```

Read Screen

```
; 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"))
```

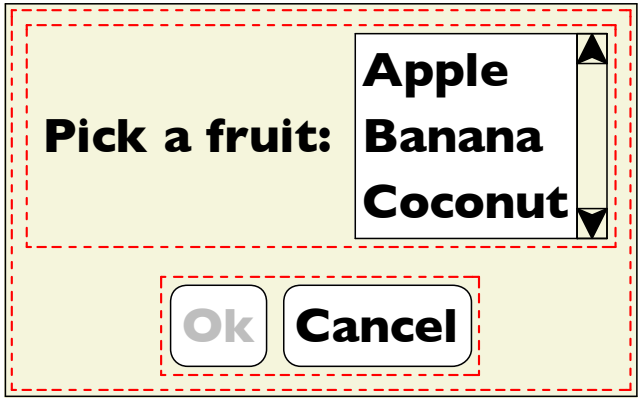
Assemblings GUIs



- 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

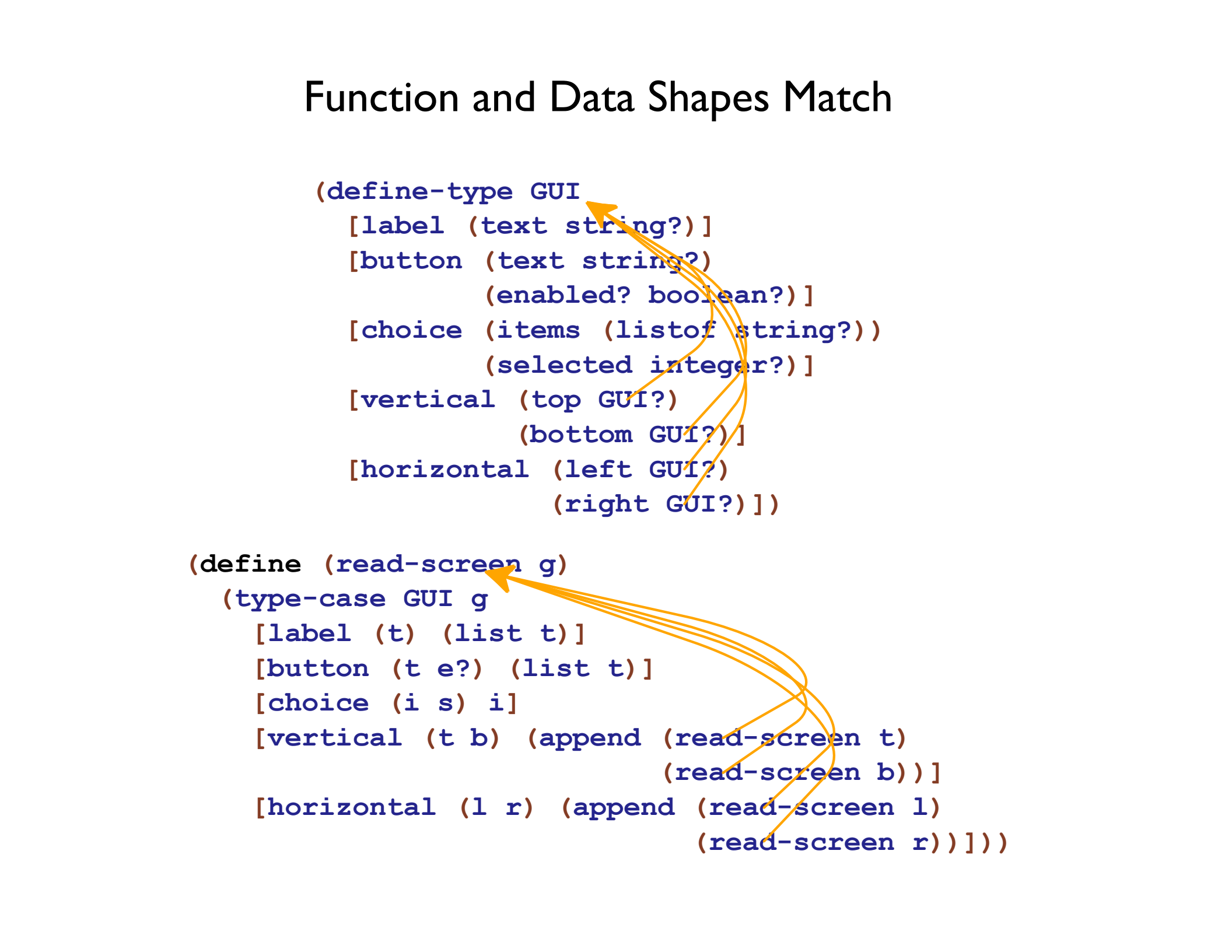
```
; read-screen : GUI -> list-of-string
(define (read-screen g)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]
    [vertical (t b) (append (read-screen t)
                            (read-screen b))]
    [horizontal (l r) (append (read-screen l)
    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

; ... earlier test cases ...
(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`
- Write examples
  - `test`
- Create a template for the implementation
  - `type-case` plus natural recursion,
    check shape!
- Finish implementation case-by-case
  - usually the interesting part, but good test cases make it less interesting (i.e., easier!)
- Run tests

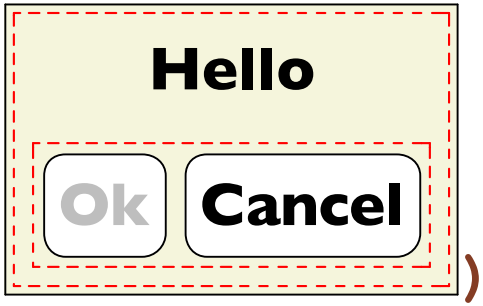
Enable Button

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

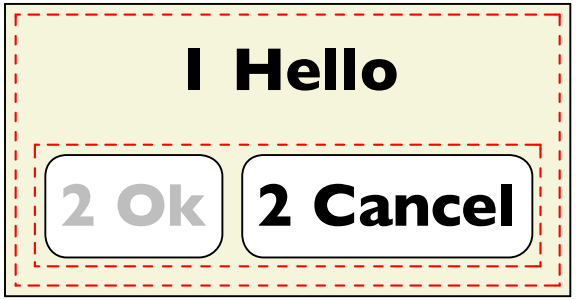
```
; enable-button : GUI string -> GUI
(define (enable-button g name)
  (type-case GUI g
    [label (t) g]
    [button (t e?) (cond
                     [(equal? t name) (button t true)]
                     [else g])]
    [choice (i s) g]
    [vertical (t b) (vertical (enable-button t name)
                              (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

Template:

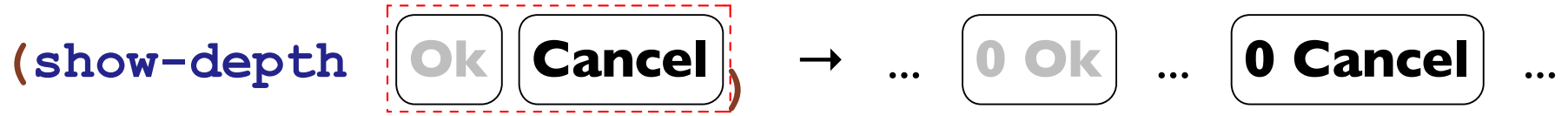
```
(define (show-depth g)
  (type-case GUI g
    [label (t) ...]
    [button (t e?) ...]
    [choice (i s) ...]
    [vertical (t b) ... (show-depth t)
                    ... (show-depth b) ...]
    [horizontal (l r) ... (show-depth l)
                      ... (show-depth r) ...]))
```

(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

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))
```

Programming With Lists

Sometimes you can use **map**, **ormap**, **for/list**, etc.

```
; has-label? : list-of-string string -> bool
(define (has-label? l s)
  (ormap (lambda (e) (string=? e s)) l))

(test (has-label? empty "Banana") false)
(test (has-label? '("Apple" "Banana") "Banana")
      true)
```

Programming With Lists

Sometimes you can use **map**, **ormap**, **for/list**, etc.

```
; has-label? : list-of-string string -> bool
(define (has-label? l s)
  (ormap (lambda (e) (string=? e s)) l))

(test (has-label? empty "Banana") false)
(test (has-label? '("Apple" "Banana") "Banana")
      true)
```

Otherwise, the general design process works for programs on lists using the following data definition:

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

Programming With Lists

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

; has-label? : list-of-string string -> bool
(define (has-label? l s)
  (cond
    [(empty? l) ...]
    [(cons? l) ... (first l)
               ... (has-label? (rest l) s) ...]))
```

Programming With Lists

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

; has-label? : list-of-string string -> bool
(define (has-label? l s)
  (cond
    [(empty? l) false]
    [(cons? l) (or (string=? (first l) s)
                   (has-label? (rest l) s))]))
```