`#lang plai`

Out meta-language of choice

The PLAI Language

- A dialect of Racket, and close cousin of the 111 student languages
- Designed to make writing meta-programs easy

- You need to get back into the 111 way of thinking / programming
- Today we'll see the new tools PLAI brings to the table

Data-Driven Design

Key Idea of 111-style programming: the shape of your data determines the shape of your functions

Step 1: define the shape of the data you're working with

- PLAI provides `define-type` to help

Step 2: write examples / test cases, following that shape

- PLAI provides `test` to help

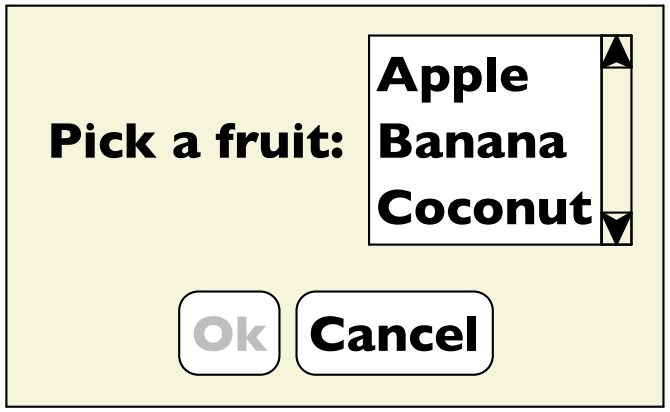
Step 3: sketch out your functions, following that shape

- PLAI provides `type-case` to help

Step 4: fill out each case

- That's the part that requires the most thinking
- But some cases will be trivial!

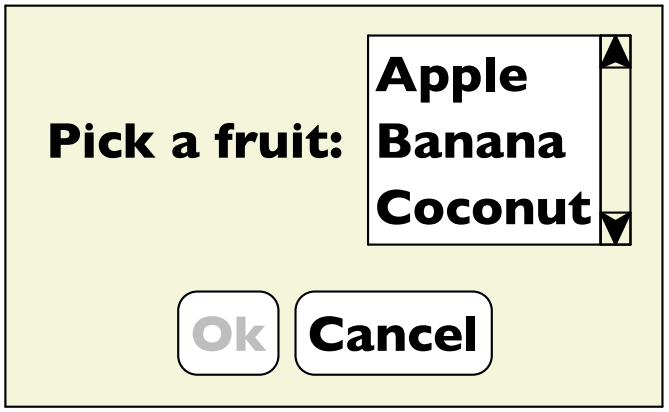
Running Example: GUIs



Possible functions:

- Read all the text on the screen
- Can I click on a given button?
- Enable a given button
- Etc.

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?)])
```

`define-type`

Declare each variant

Declare data each needs to keep track of

And specify what kind of data for each

Read Screen

Produce a list with all the text we find in the given GUI

`test` compares a computed value with an expected value

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

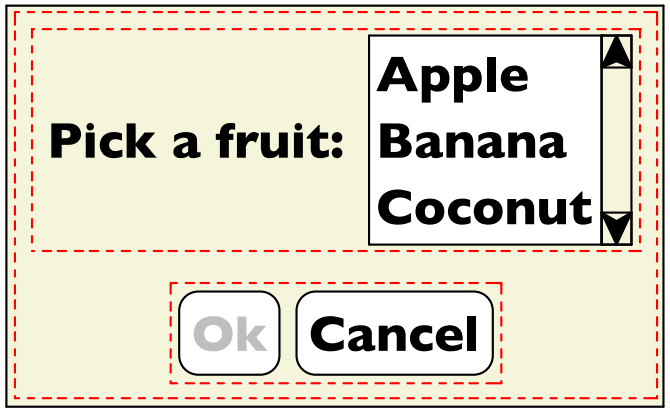
Read Screen

Produce a list with all the text we find in the given GUI

`type-case` dispatches on the possible variants, and introduces local variables for each of their fields

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

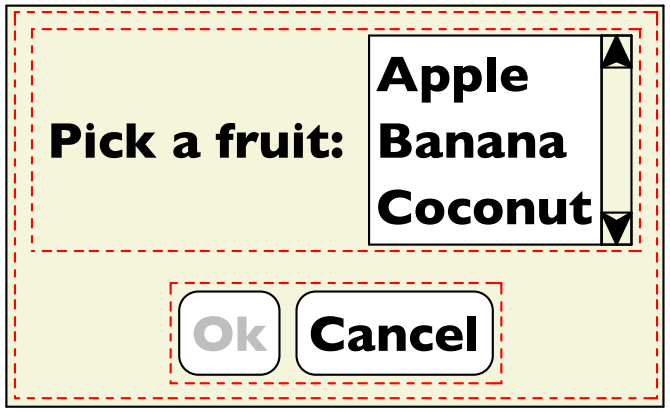
Assemblings GUIs



- 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, take 2

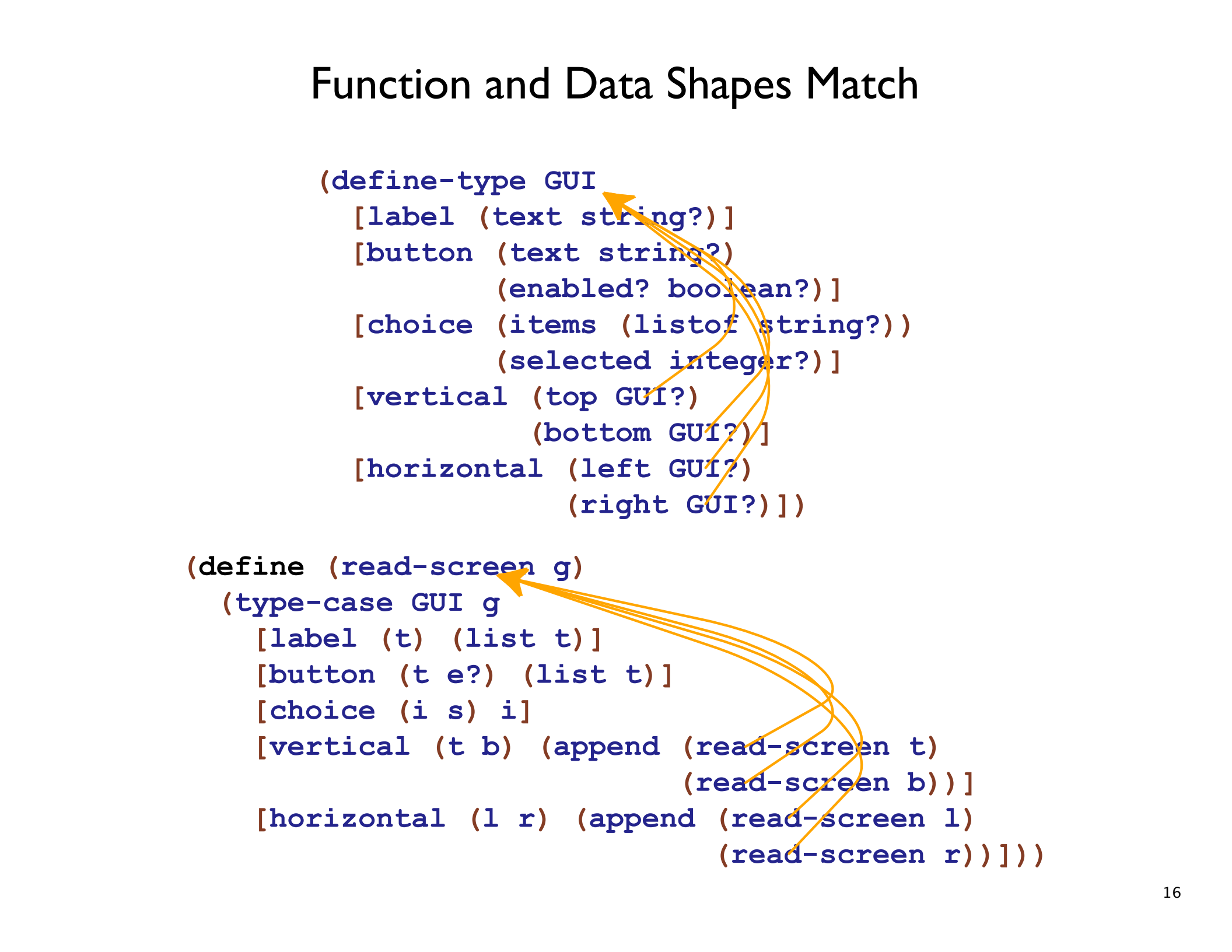
```
; read-screen : GUI? -> (listof string?)
(define (read-screen g)
  (type-case GUI g
    [label (t) (list t)]
    [button (t e?) (list t)]
    [choice (i s) i]
    [vertical (t b) (append (read-screen t)
                            (read-screen b))]
    [horizontal (l r) (append (read-screen l)
 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

; ... earlier test cases ...
(test (read-screen 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))]))
```



Further Techniques

That was the basic way of designing our functions, which will work most of the time.

But sometimes we'll need slightly different function shapes.

Two examples:

- Passing information along
- Passing accumulators

Passing Information Along

We need the button name in the leaves of the tree.

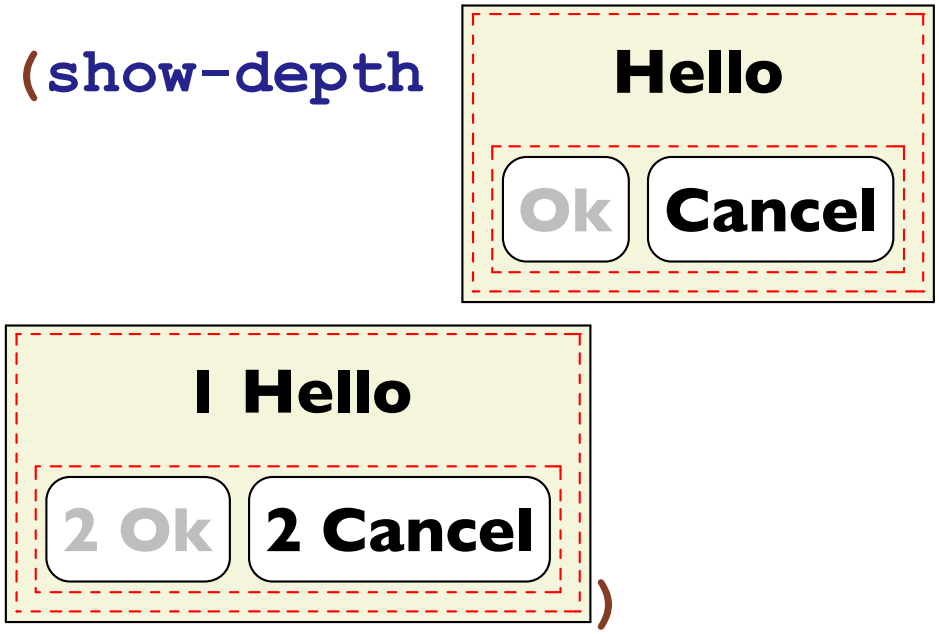
We recur on **`g`** and **`name`** follows along unchanged.

```
; 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))))
```

Passing Accumulators

Edit each label to add depth in the GUI tree

```
(test (show-depth
```



Passing Accumulators

The **n** argument is an *accumulator*. We update it as we go deeper.

```
; show-depth-at : GUI? integer? -> 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))
```