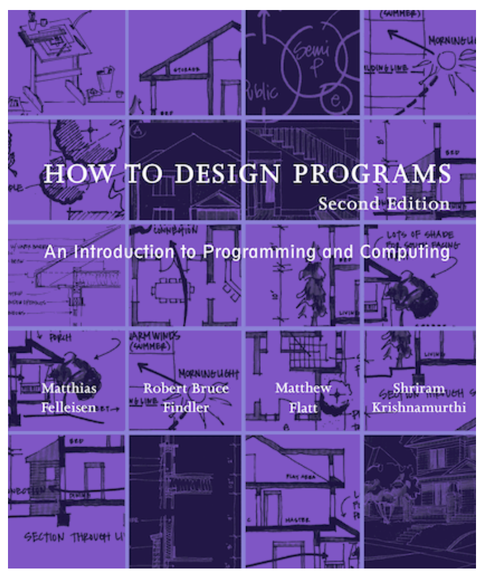
How to Design Programs

using Shplait



`http://www.htdp.org`

How to Design Programs

- Determine the **representation**
  - `type`, if needed
- Write **examples**
  - `check`
- Create a **template** for the implementation
  - `match`, 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

```
Int

eat_cookie :: Int -> Int
```

Examples

```
Int

eat_cookie :: Int -> Int

check:
       ~is
```

Examples

```
Int

eat_cookie :: Int -> Int

check: eat_cookie(   )
       ~is
```

Examples

```
Int

eat_cookie :: Int -> Int

check: eat_cookie(10)
       ~is
```

Examples

```
Int

eat_cookie :: Int -> Int

check: eat_cookie(10)
       ~is 9
```

Examples

```
Int

eat_cookie :: Int -> Int

check: eat_cookie(10)
       ~is 9
check: eat_cookie(1)
       ~is 0
check: eat_cookie(0)
       ~is 0
```

Template

```
Int

eat_cookie :: Int -> Int

fun eat_cookie(n :: Int):
  .... n ....
```

Body

```
Int

eat_cookie :: Int -> Int

fun eat_cookie(n :: Int):
  .... n ....



check: eat_cookie(10)
         ~is 9
check: eat_cookie(1)
         ~is 0
check: eat_cookie(0)
         ~is 0
```

Body

```
Int

eat_cookie :: Int -> Int

fun eat_cookie(n :: Int):
  n - 1



check: eat_cookie(10)
       ~is 9
check: eat_cookie(1)
       ~is 0
check: eat_cookie(0)
       ~is 0
```

Body

```
Int

eat_cookie :: Int -> Int

fun eat_cookie(n :: Int):
  if n > 0
  | n - 1
  | 0

check: eat_cookie(10)
       ~is 9
check: eat_cookie(1)
       ~is 0
check: eat_cookie(0)
       ~is 0
```

Test

```
Int

eat_cookie :: Int -> Int

fun eat_cookie(n :: Int):
  if n > 0
  | n - 1
  | 0

check: eat_cookie(10)
       ~is 9
check: eat_cookie(1)
       ~is 0
check: eat_cookie(0)
       ~is 0
```

Representation

- Track a position on the screen

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn
```

Examples

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

check: flip(           )
       ~is
```

Examples

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

check: flip(posn(1, 17))
       ~is
```

Examples

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

check: flip(posn(1, 17))
       ~is posn(17, 1)
```

Examples

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

check: flip(posn(1, 17))
       ~is posn(17, 1)
check: flip(posn(-3, 4))
       ~is posn(4, -3)
```

Template

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

fun flip(p :: Posn):
  .... posn.x(p)
  .... posn.y(p) ....
```

or

```
fun flip(p :: Posn):
  match p
  | posn(x, y): .... x .... y ....
```

Body

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

fun flip(p :: Posn):
  match p
  | posn(x, y): .... x .... y ....

check: flip(posn(1, 17))
       ~is posn(17, 1)
check: flip(posn(-3, 4))
       ~is posn(4, -3)
```

Body

```
type Posn
| posn(x :: Int,
       y :: Int)

flip :: Posn -> Posn

fun flip(p :: Posn):
  match p
  | posn(x, y): posn(y, x)

check: flip(posn(1, 17))
       ~is posn(17, 1)
check: flip(posn(-3, 4))
       ~is posn(4, -3)
```

Representation

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

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean
```

Examples

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

check: is_ant_home(ant(posn(0, 0), 1))
       ~is #true
check: is_ant_home(ant(posn(5, 10), 1))
       ~is #false
```

Template

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

fun is_ant_home(a :: Ant):
  match a
  | ant(loc, wgt):
      .... loc ....
      .... wgt ....
```

Template

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

fun is_ant_home(a :: Ant):
  match a
  | ant(loc, wgt):
      .... is_home(loc) ....
      .... wgt ....

fun is_home(p :: Posn):
  match p
  | posn(x, y): .... x .... y ....
```

Body

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

fun is_ant_home(a :: Ant):
  match a
  | ant(loc, wgt):
      .... is_home(loc) ....
      .... wgt ....

fun is_home(p :: Posn):
  match p
  | posn(x, y): .... x .... y ....
```

Body

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

fun is_ant_home(a :: Ant):
  match a
  | ant(loc, wgt): is_home(loc)

fun is_home(p :: Posn):
  match p
  | posn(x, y): .... x .... y ....
```

Body

```
type Ant
| ant(location :: Posn,
      weight :: Int)

is_ant_home :: Ant -> Boolean

fun is_ant_home(a :: Ant):
  match a
  | ant(loc, wgt): is_home(loc)

fun is_home(p :: Posn):
  match p
  | posn(x, y): x == 0 && y == 0
```

Representation

- Track an animal, which is a tiger or a snake

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

```
check: is_heavy_animal(                    )
       ~is
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean

check: is_heavy_animal(tiger(#'orange, 14))
       ~is
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

```
check: is_heavy_animal(tiger(#'orange, 14))
       ~is #true
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

```
check: is_heavy_animal(tiger(#'orange, 14))
       ~is #true
check: is_heavy_animal(snake(#'green, 10, "rats"))
       ~is
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

```
check: is_heavy_animal(tiger(#'orange, 14))
         ~is #true
check: is_heavy_animal(snake(#'green, 10, "rats"))
         ~is #true
```

Examples

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean
```

```
check: is_heavy_animal(tiger(#'orange, 14))
       ~is #true
check: is_heavy_animal(snake(#'green, 10, "rats"))
       ~is #true
check: is_heavy_animal(snake(#'yellow, 8, "cake"))
       ~is #false
```

Template

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean

fun is_heavy_animal(a :: Animal):
  match a
  | tiger(c, sc):
      .... c .... sc ....
  | snake(c, w, f):
      .... c .... w ....
      .... f ....
```

Body

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean

fun is_heavy_animal(a :: Animal):
  match a
  | tiger(c, sc):
      .... c .... sc ....
  | snake(c, w, f):
      .... c .... w ....
      .... f ....
```

Body

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean

fun is_heavy_animal(a :: Animal):
  match a
   | tiger(c, sc): #true
   | snake(c, w, f):
       .... c .... w ....
       .... f ....
```

Body

```
type Animal
| tiger(color :: Symbol,
        stripe_count :: Int)
| snake(color :: Symbol,
        weight :: Int,
        food :: String)

is_heavy_animal :: Animal -> Boolean

fun is_heavy_animal(a :: Animal):
  match a
   | tiger(c, sc): #true
   | snake(c, w, f): w >= 10
```

Representation

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

```
Listof(Int)
```

Representation

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

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)
```

Examples

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

Examples

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

check: feed_fish([])
       ~is []
check: feed_fish([1, 2, 3])
       ~is [2, 3, 4]
```

Template

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: ....
  | cons(n, rst_lon): ....
```

Template

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
   | []: ....
   | cons(n, rst_lon):
       .... n ....
       .... rst_lon ....
```

Template

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: ....
  | cons(n, rst_lon):
      .... n ....
      .... feed_fish(rst_lon) ....
```

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: ....
  | cons(n, rst_lon):
      .... n ....
      .... feed_fish(rst_lon) ....
```

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: ....
  | cons(n, rst_lon):
      .... n ....
      .... feed_fish(rst_lon) ....
check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: []
  | cons(n, rst_lon):
      .... n ....
      .... feed_fish(rst_lon) ....
check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: []
  | cons(n, rst_lon):
      .... 1 + n ....
      .... feed_fish(rst_lon) ....
check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
   | []: []
   | cons(n, rst_lon):
       .... 1 + n ....
       .... feed_fish(rst_lon) ....
check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

rst_lon

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: []
  | cons(n, rst_lon):
      .... 1 + n ....
      .... feed_fish(rst_lon) ....
check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

rst_lon

feed_fish(rst_lon)

Body

```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))

feed_fish :: Listof(Int) -> Listof(Int)

fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: []
  | cons(n, rst_lon):
      cons(1 + n,
           feed_fish(rst_lon))

check: feed_fish([])
       ~is []
check: feed_fish(cons(1, cons(2, cons(3, []))))
       ~is cons(2, cons(3, cons(4, [])))
```

rst_lon

feed_fish(rst_lon)

Implementation Matches Data

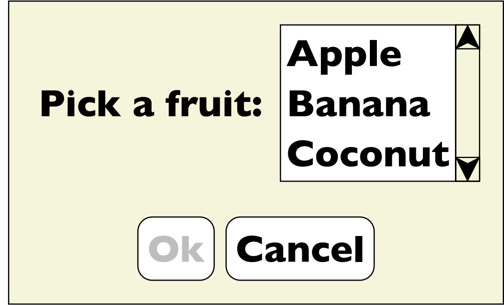
```
type Listof(Int)
| []
| cons(n :: Int,
       rst :: Listof(Int))
```

```
fun feed_fish(lon :: Listof(Int)):
  match lon
  | []: ....
  | cons(n, rst_lon): .... n
                      .... feed_fish(rst_lon) ....
```

How to Design Programs

More Examples

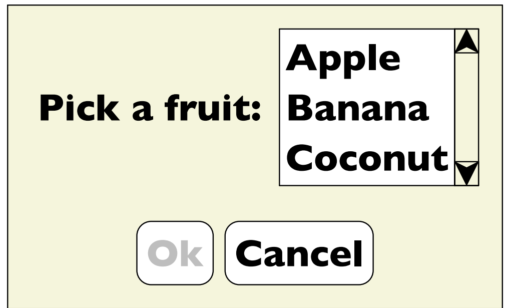
GUIs



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

```
type GUI
| label(text :: String)
| button(text :: String,
         is_enabled :: Boolean)
| choice(items :: Listof(String),
         selected :: Int)
```

Read Screen

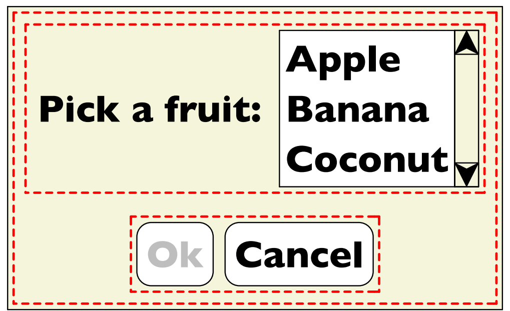
- Implement **read_screen**, which takes a GUI and returns a list of strings for all the GUI element labels

Read Screen

```
fun read_screen(g :: GUI) :: Listof(String):
  match g
  | label(t): [t]
  | button(t, e): [t]
  | choice(i, s): i

check: read_screen(label("Hi"))
       ~is ["Hi"]
check: read_screen(button("Ok", #true))
       ~is ["Ok"]
check: read_screen(choice(["Apple", "Banana"],
                          0))
       ~is ["Apple", "Banana"]
```

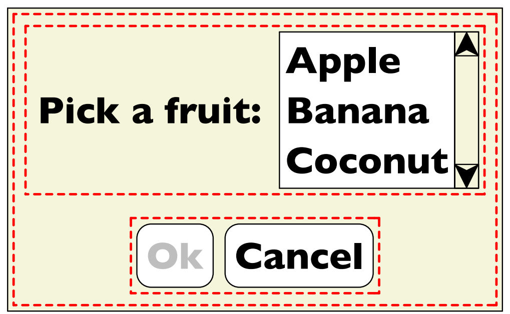
Assembling GUIs



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

```
type GUI
| label(text :: String)
| button(text :: String,
         is_enabled :: Boolean)
| choice(items :: Listof(String),
         selected :: Int)
| vertical(top :: GUI,
           bottom :: GUI)
| horizontal(left :: GUI,
             right :: GUI)
```

Assembling GUIs



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

```
def 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

```
fun read_screen(g :: GUI) :: Listof(String):
  match g
  | label(t): [t]
  | button(t, e): [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))

....
check: read_screen(gui1)
       ~is ["Pick a fruit:",
            "Apple", "Banana", "Coconut",
            "Ok", "Cancel"]
```

Function and Data Shapes Match

```
type GUI
| label(text :: String)
| button(text :: String,
         is_enabled :: Boolean)
| choice(items :: Listof(String),
         selected :: Int)
| vertical(top :: GUI,
           bottom :: GUI)
| horizontal(left :: GUI,
             right :: GUI)

fun read_screen(g :: GUI) :: Listof(String):
  match g
  | label(t): [t]
  | button(t, e): [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
  - `type`, maybe
- Write examples
  - `check`
- Create a template for the implementation
  - `match` plus natural recursion, **check shape!**
- Finish body implementation case-by-case
  - *usually the interesting part*
- Run tests

Enable Button

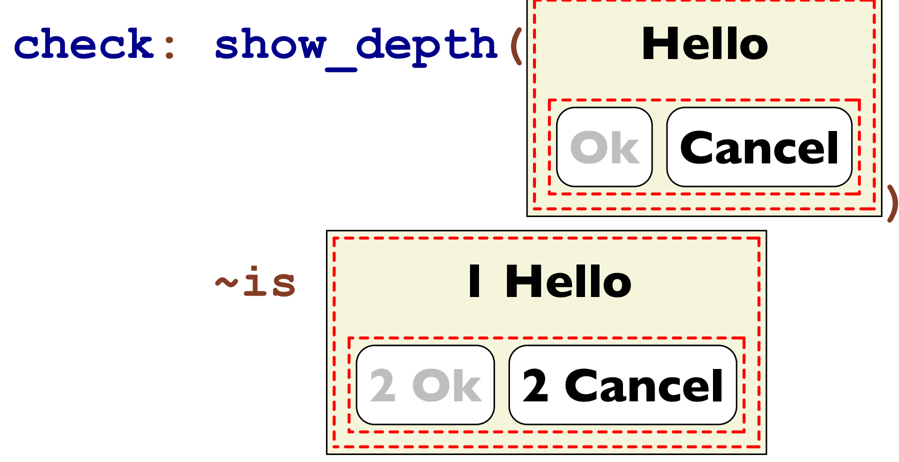
- Implement **`enable_button`**, which takes a GUI and a string and enables the button whose name matches the string

Enable Button

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

```
fun enable_button(g :: GUI, name :: String) :: GUI:
  match g
  | label(t): g
  | button(t, e): cond
                  | 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))
....
check: enable_button(gui1, "Ok")
       ~is vertical(horizontal(label("Pick a fruit:"),
                               choice(["Apple", "Banana", "Coconut"], 0)),
                    horizontal(button("Ok", #true),
                               button("Cancel", #true)))
```

Show Depth



Show Depth

Template:

```
fun show_depth(g :: GUI) :: GUI:
  match 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:

```
fun show_depth(g :: GUI) :: GUI:
  match 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 ....

Show Depth

Template:

```
fun show_depth(g :: GUI) :: GUI:
  match 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 `n` argument is an *accumulator*:

```
fun show_depth_at(g :: GUI, n :: Int) :: GUI:
  match 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))

fun show_depth(g :: GUI) :: GUI:
  show_depth_at(g, 0)
```

How to Design Programs

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