How to Design Programs

How to (in Scheme):

• represent data
  ○ variants
  ○ trees and lists

• write functions that process the data

See also

http://www.htdp.org
Running Example: GUIs

Pick a fruit:
- Apple
- Banana
- Coconut

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 GUls

- 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 guil
  (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))])

... 
(test guil
  ('("Pick a fruit:
    "Apple" "Banana" "Coconut"
    "Ok" "Cancel"))

(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
  - the is usually the interesting part

- Run tests
Enable Button

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

```scheme
; 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

\[(\text{test } (\text{show-depth } 1 \text{ Hello})\text{ Ok } \text{ Cancel})\]
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) ...]))
```
Show Depth

Template:

\[
\begin{align*}
& (define \ (show-depth \ g) \\
& \hspace{1cm} (type-case \ GUI \ g) \\
& \hspace{2cm} [\text{label} \ (t) \ ...] \\
& \hspace{2cm} [\text{button} \ (t \ e?) \ ...] \\
& \hspace{2cm} [\text{choice} \ (i \ s) \ ...] \\
& \hspace{2cm} [\text{vertical} \ (t \ b) \ ... (show-depth \ t) \\
& \hspace{3.5cm} ... (show-depth \ b) \ ...] \\
& \hspace{2cm} [\text{horizontal} \ (l \ r) \ ... (show-depth \ l) \\
& \hspace{3.5cm} ... (show-depth \ r) \ ...])] \\
\end{align*}
\]

\[
(show-depth \ \boxed{Ok}) \rightarrow \ 0 \ \boxed{Ok}
\]
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) \rightarrow \ldots 0 \text{Ok} \ldots 0 \text{Cancel} \ldots
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 \texttt{n} argument is an \textit{accumulator}:

\begin{verbatim}
; 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))
\end{verbatim}
Sometimes you can use map, ormap, 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)
Sometimes you can use `map`, `ormap`, etc.

\[
\text{; has-label? : list-of-string string -> bool}
\]
\[
\text{(define (has-label? l s)}
\]
\[
\text{ (ormap (lambda (e) (string=? e s)) l))}
\]
\[
\text{(test (has-label? empty "Banana") false)}
\]
\[
\text{(test (has-label? '("Apple" "Banana") "Banana") true)}
\]

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

\[
\text{; A list-of-string is either}
\]
\[
\text{; - empty}
\]
\[
\text{; - (cons string list-of-string)}
\]
; 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) ...]))
; 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))]]))