Abstractions & First Class Functions

- We have seen that functions can be passed (as parameters) to other functions:

```
(map integer? '(1 fred 2 sally))
```

- Scheme functions can also be produced by functions!
  - we can build functions that are "function factories".
  - we should think of functions as being data that can be created, passed and used like any other data.

A function that creates a function.

```
(define (make_add_func x)
  (local ((define (x-adder y) (+ x y)))
    x-adder))

(define add5 (make_add_func 5))
(add5 10) => 15
(add5 21) => 26
(define add2k (make_add_func 2048))
```

Another Example: filter functions

- Remember filter1?

```
(define (filter1 rel-op alon t)
  (cond
    [(empty? alon) empty]
    [else (cond
    [(rel-op (first alon) t)
     (cons (first alon)
           (filter1 rel-op (rest alon) t))]
    [else
     (filter1 rel-op (rest alon) t)])]))
```

Using `filter1` to be `above/below`

- To act like `below`, we give it `rel-op <`:

```
> (filter1 < '(8 1 7 253 49 26 4) 17)
(list 8 1 7 4)
```

- To act like `above`, we give it `rel-op >`:

```
> (filter1 > '(8 1 7 253 49 26 4) 17)
(list 253 49 26)
```

Filter-creator

```
(define (filter-creator rel-op)
  (local
   ((define (filter1 alon t)
      (cond
       [(empty? alon) empty]
       [else
        (cond
         [(rel-op (first alon) t)
          (cons (first alon)
                (filter1 (rest alon) t))]
         [else
          (filter1  (rest alon) t)])])))
   filter1))
```

Using `filter-creator`

```
(define above (filter-creator > ))
(above '(1 7 9 27 56 19) 20) =>
'(27 56)

(define below (filter-creator < ))
(below  '(1 7 9 27 56 19) 20) =>
'(1 7 8 19)
```

Template for a function-creating-function

```
(define (fcf p1 p2 ...)
  (local
     ((define (afunc params ...)
         uses p1 p2 ...))
   afunc))
```

Exercise

- We want bunch of functions like `first`, `second`, `third`, etc.
- Write a function named nth-creator that consumes an integer n and produces a function that will select the nth element of a list.

```
(define tenth (nth-creator 10))

(tenth '(1 2 3 4 5 6 7 8 9 10 11 12 13))
  => 10
```

GUI Design

- Graphical User Interface software has evolved over many years to what is called

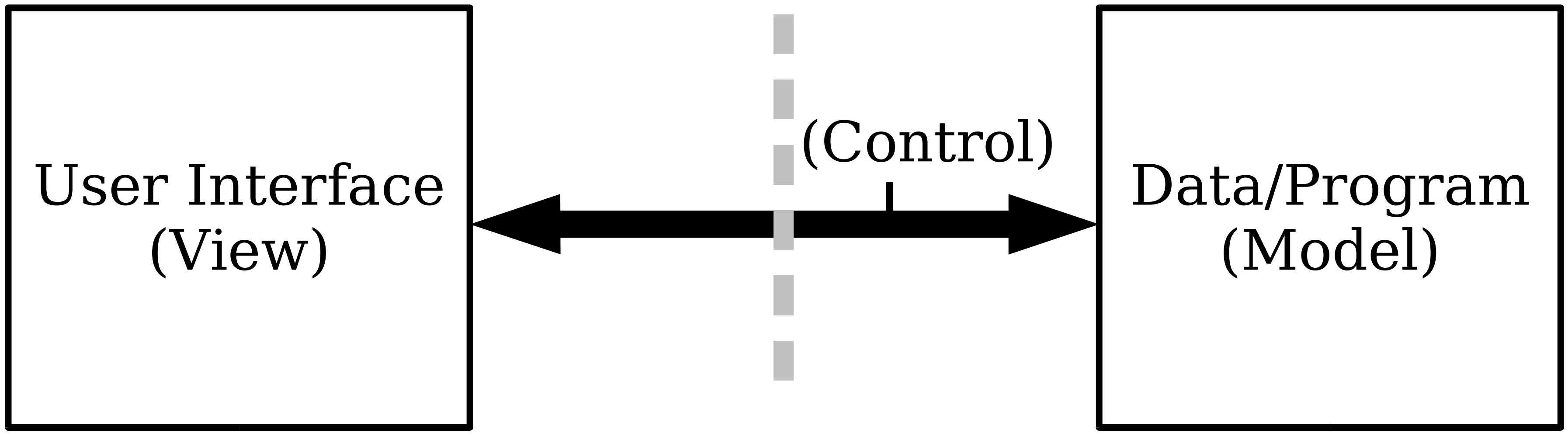
  "Model-View-Control" architecture

- This architecture is supported by just many GUI building libraries and support tools.
- MVC makes heavy use of functions as "first class values".
  - it relies on the ability to pass functions as parameters.

Model-View-Control

- Model: the data being manipulated, or activities being controlled.
- View: the presentation of the data to the user.
  - can be independent of the actual way in which the data is represented or generated internally.
- Control: the glue. Provides mechanisms for:
  - associating GUI events with changing the data.
  - associating changes to data with GUI updates.

MVC Overview



Teachpack `gui.ss`

- Simple compared to complete gui libraries
  - enough to get a feel for GUI programming and MVC architecture.
  - limited number/type of components
  - no real control over layout
    - real libraries include *layout managers*
  - limited types of events

gui-item Data Definition

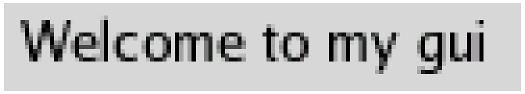
A gui-item is either:

- a button: `(make-button string (X ->  true))`
- a textbox: `(make-text string)`
- a menu: `(make-choices (listof string))`
- a message: `(make-message string)`

GUI components: message

- Create a message (a string):

```
(make-message "Welcome to my gui")
```


Welcome to my gui

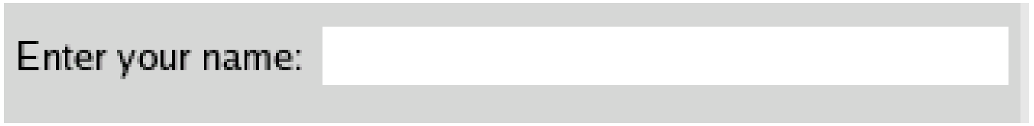
- Message can be changed later:

```
(define m1 (make-message  "blah"))
(draw-message m1 "get rid of blah")
```

GUI components: text

- Create a textbox with label:

```
(make-text "Enter your name:")
```



- Contents can be retrieved:

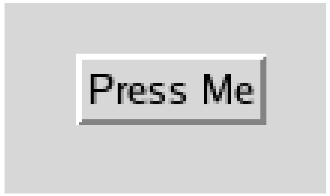
```
(define t1 (make-text "Name:"))


         (text-contents t1)
```

GUI components: button

- Create a button with a *callback function*:

```
(make-button "Press Me" hide-window))
```



- Callback function is called when button is pressed.

GUI components: window

- Create a window that contains a number of gui items.

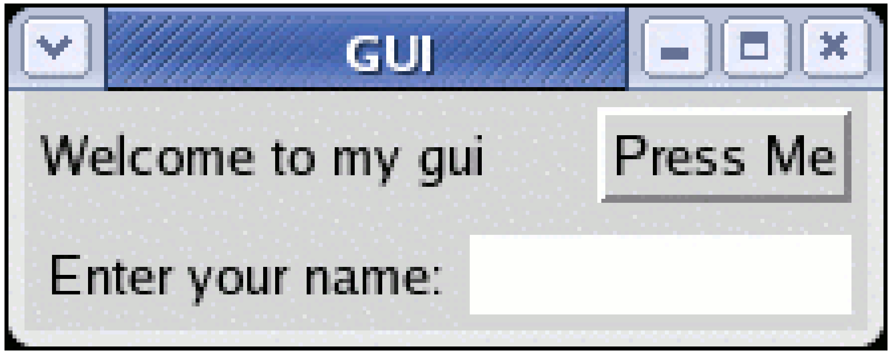
```
(create-window
    (list
        (list item1 item2 ...)
        (list item3 item4 ...)))
```

Each item is a gui-item

Each list is a row in the window

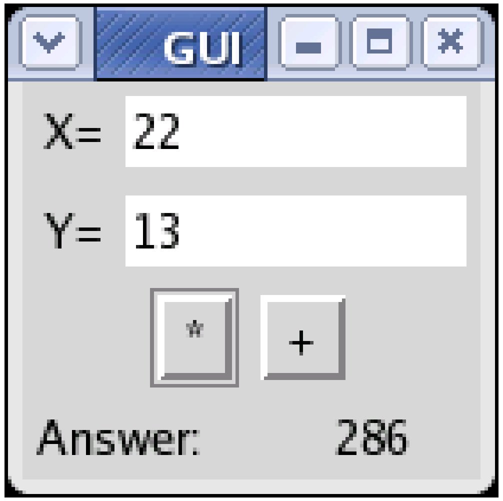
Window example

```
(create-window
  (list
    (list
      (make-message "Welcome to my gui")
      (make-button "Press Me" hide-window))
    (list (make-text "Enter your name:"))))
```



Simple Calculator Example

- User can enter two numbers
  - actually textbox allows entry of only strings!
- Buttons to indicate multiplication and addition.
- When button is pressed the answer is displayed as a message.



Item definitions

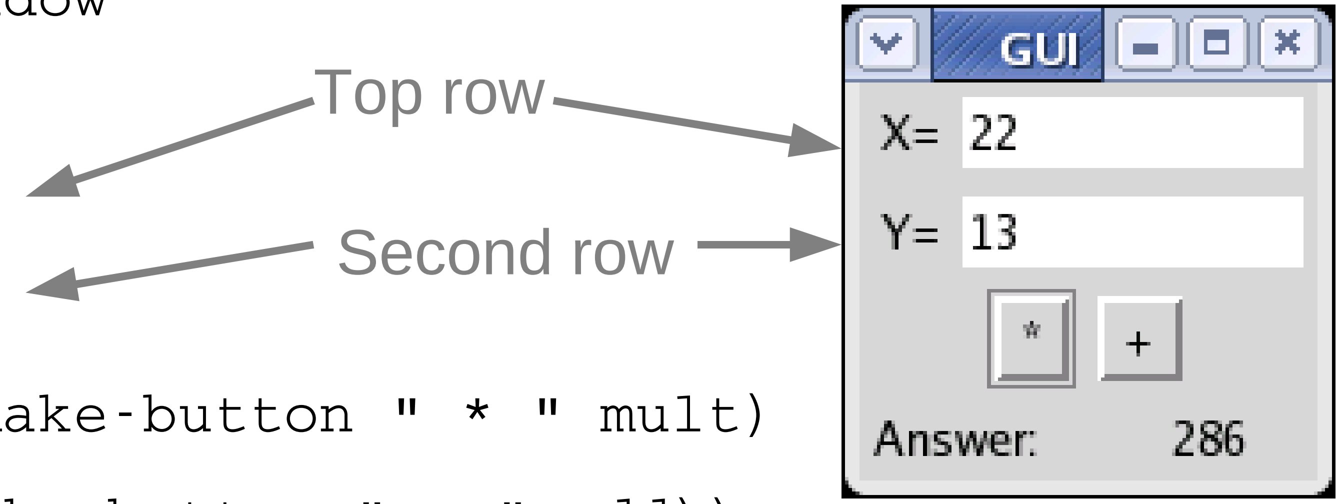
- Some items need to have names so we can get at them later:

```
;; define some GUI items we will use
;; x and y are textboxes
(define x (make-text "X="))
(define y (make-text "Y="))

;; answer is a message (where we draw the answer)
(define answer (make-message "---"))
```

Window Creation

```
(create-window
 (list
  (list  x)
  (list  y)
  (list  (make-button " * " mult)
         (make-button " + " add))
  (list  (make-message "Answer: ") answer)))
```



X= 22

Y= 13

Answer: 286

Button handlers

- Functions that accept a single parameter
  - the parameter passed represents the *event* that caused the function to be called. We can ignore this for now.
- We need two:
  - `mult` will extract numbers from the textboxes, multiple the numbers, convert result to string and draw the answer in the message `answer`.
  - `add` will do something similar...

Strings and Numbers

- Textboxes and messages are strings.
- We need to deal with numbers.

```
(string->number "24") => 24

(number->string 37) => "37"
```

`mult` function

```
;; mult is called when the user clicks on the
;;   * button
(define (mult e)
  (draw-message
   answer
   (number->string
    (* (string->number (text-contents x))
       (string->number (text-contents y))))))
```

`add` function

```
;; add is called when the user clicks on
;;    the + button
(define (add e)
  (draw-message
   answer
   (number->string
    (+ (string->number (text-contents x))
       (string->number (text-contents y))))))
```

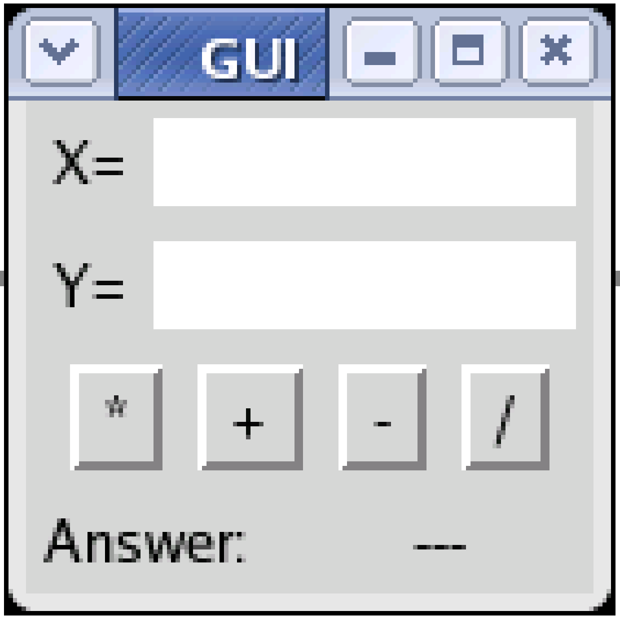
`mult` and `add`

- Very similar functions
  - possibility of replacing with a more general function?
- We could write a single function that needs a + or * passed to it,
  - but we don't call the function! The gui code calls `mult` and `add` for us.
  - we don't have any way to tell the gui code to pass a parameter – it always passed the event only.

How about using a function that creates functions?

```
(define (make-button-handler op)
  (local
   ((define (f e)
       (draw-message
        answer
         (number->string
          (op (string->number (text-contents x))
              (string->number (text-contents y)))))))
   f))
```

New version



```
(create-window
 (list
  (list  x)
  (list  y)
  (list  (make-button " * " (make-button-handler *))
         (make-button " + " (make-button-handler +))
         (make-button " - " (make-button-handler -))
         (make-button " / " (make-button-handler /)))
  (list  (make-message "Answer: ") answer)))
```

Exercise

- Create a GUI so that the user can enter a number indicating a temperature in degrees Fahrenheit.
- When a button is pressed the GUI displays the temperature in Celsius (in a message).

Celsius = (Fahrenheit-32) * (5/9)