Calculator

Run
Adding Machine?

![GUI](image.png)

Run
Adding Machine?

```
(require htdp/gui)

(define TOTAL 0)

(define total-message
  (make-message (number->string TOTAL)))

(define amount-text
  (make-text "Amount"))

(define add-button
  (make-button "+
    (lambda (evt)
      (add-to-total
        (string->number (text-contents amount-text))))))

; add-to-total : num -> true
(define (add-to-total amt)
  (local ((define new-total (+ TOTAL amt)))
    (draw-message total-message total-message (number->string new-total))))

(create-window (list (list total-message)
      (list amount-text)
      (list add-button)))
```
Why the Adding Machine Doesn’t Work

(define (add-to-total amt)
  (local [(define new-total (+ TOTAL amt))]
    (draw-message total-message (number->string new-total))))

• Every time we have a new amt, it’s added to the same original TOTAL

• The new total is drawn on the screen, then forgotten

• The GUI library doesn’t keep a “world” for us

We need a side channel to save new-total
In Advanced Student Language:

(set! TOTAL 17)

changes the value of TOTAL to 17

• “Constant” definitions are no longer constant — they are variable definitions

• A set! expression is called an assignment

• The value of TOTAL is the state of the program
Evaluating set!

```
(define TOTAL 0)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(add-amt 1)
(add-amt 2)

→

(define TOTAL 0)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(set! TOTAL (+ TOTAL 1))
(add-amt 2)
```
Evaluating set!

(define TOTAL 0)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(set! TOTAL (+ TOTAL 1))
(add-amt 2)

→

(define TOTAL 0)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(set! TOTAL (+ 0 1))
(add-amt 2)
Evaluating set!

(define TOTAL 0)
(define (add- amt amt)
  (set! TOTAL (+ TOTAL amt)))
(set! TOTAL (+ 0 1))
(add- amt 2)

→

(define TOTAL 0)
(define (add- amt amt)
  (set! TOTAL (+ TOTAL amt)))
(set! TOTAL 1)
(add- amt 2)
Evaluating set!

(define TOTAL 0)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(set! TOTAL 1)
(add-amt 2)

\[\text{\rightarrow}\]

(define TOTAL 1)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(void)
(add-amt 2)

To evaluate \textbf{set!}, change a definition and produce
(void)
Evaluating set!

(define TOTAL 1)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(add-amt 2)

→

(define TOTAL 1)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL (+ TOTAL 2))
Evaluating set!

(define TOTAL 1)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL (+ TOTAL 2))

→

(define TOTAL 1)
(define (add-amt amt)
    (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL (+ 1 2))
Evaluating set!

```
(define TOTAL 1)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL (+ 1 2))

→

(define TOTAL 1)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL 3)
```

It’s important that a variable name is not replaced by its value until the value is needed.
Evaluating set!

```
(define TOTAL 1)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(set! TOTAL 3)

→

(define TOTAL 3)
(define (add-amt amt)
  (set! TOTAL (+ TOTAL amt)))
(void)
(void)
```
Making the Adder Remember Totals

(define (add-to-total amt)
  (local [(define new-total (+ TOTAL amt))]
    ; How do we combine two actions?
    ...
    (set! TOTAL new-total)
    (draw-message total-message (number->string new-total))
    ...))
Making the Adder Remember Totals

```
(define (add-to-total amt)
  (local ((define new-total (+ TOTAL amt)))
    (begin
      (set! TOTAL new-total)
      (draw-message total-message (number->string new-total)))))
```

Also new in **Advanced**: the **begin** form

The **begin** form

- Evaluates its first expression
- Throws away the result
- Goes away when only one expression is left

**begin** works with any number of expressions
Evaluating begin

```
(define TOTAL 3)
(define (running-total amt)
  (begin
    (set! TOTAL (+ TOTAL amt))
    TOTAL))
(running-total 10)

→

(define TOTAL 3)
...
(begin
  (set! TOTAL (+ TOTAL 10))
  TOTAL)
```
Evaluating begin

(define TOTAL 3)
...
(begin
  (set! TOTAL (+ TOTAL 10))
  TOTAL)

→

(define TOTAL 3)
...
(begin
  (set! TOTAL (+ 3 10))
  TOTAL)
Evaluating begin

(define TOTAL 3)
...
(begin
  (set! TOTAL (+ 3 10))
  TOTAL)

→

(define TOTAL 3)
...
(begin
  (set! TOTAL 13)
  TOTAL)
Evaluating begin

(define TOTAL 3)
...
(begin
  (set! TOTAL 13)
  TOTAL)

→

(define TOTAL 13)
...
(begin
  (void)
  TOTAL)
Evaluating begin

(define TOTAL 13)
...
(begin
  (void)
  TOTAL)

→

(define TOTAL 13)
...
(begin
  TOTAL)
Evaluating begin

```
(define TOTAL 13)
...
(begin
  TOTAL)
→
(define TOTAL 13)
...
TOTAL
```
Evaluating begin

(define TOTAL 13)
...
TOTAL

→

(define TOTAL 13)
...
13
More Calculator Buttons

8
Amount 4

Run
Implementing More Calculator Buttons

...  

; op-button : string (num num -> num) -> button  
(define (op-button label OP)  
    (make-button label  
        (lambda (evt)  
            (change-total  
                OP  
                (string->number (text-contents amount-text))))))  

; change-total : (num num -> num) num -> true  
(define (change-total OP amt)  
    (local ((define new-total (OP TOTAL amt)))  
        (begin  
            (set! TOTAL new-total)  
            (draw-message total-message (number->string new-total))))))  

(create-window (list (list total-message)  
    (list amount-text)  
    (list (op-button "+" +)  
        (op-button "-" -)  
        (op-button "*" *)  
        (op-button "/" /))))
The Digit Buttons

Now two pieces of state:

• The running total
• The number we’re typing, so far
Implementing Digit Buttons

... 
(define WORKING 0)

; digit-button : num -> button
(define (digit-button n)
  (make-button (number->string n)
    (lambda (evt)
      (add-digit n)))))

; add-digit : num -> true
(define (add-digit n)
  (begin
    (set! WORKING (+ n (* WORKING 10)))
    (draw-message total-message (number->string WORKING))))

; change-total : (num num -> num) num -> true
(define (change-total OP amt)
  (local ((define new-total (OP TOTAL amt)))
    (begin
      (set! TOTAL new-total)
      (set! WORKING 0)
      (set! WORKING 0)
      (draw-message total-message (number->string new-total)))))))
...


Infix Operations

![Calculator GUI](image)

A normal calculator uses infix (algebra-like) order.

New piece of state:

- The operation to perform when the number is ready.
Implementing Infix Operations

... (define PREV-OP +)

; op-button : string (num num -> num) -> button
(define (op-button label OP)
    (make-button label
        (lambda (evt)
            (begin
                (change-total PREV-OP WORKING)
                (set! PREV-OP OP)
                true)))))
...

(create-window (list (list total-message)
    (map digit-button '(7 8 9))
    (map digit-button '(4 5 6))
    (map digit-button '(1 2 3))
    (map digit-button '(0))
    (list (op-button "+" +)
        (op-button "-" -)
        (op-button "+" *
        (op-button "/" /
        (op-button "=" (lambda (tot new) new))))))
Multiple Calculators

Use **local** to create separate **TOTALs**
Implementing Multiple Calculators

(define (make-calculator)
  (local ((define TOTAL 0)
           (define WORKING 0)
           ...

  (create-window
    (list (list total-message)
           (map digit-button '(7 8 9))
           (map digit-button '(4 5 6))
           (map digit-button '(1 2 3))
           (map digit-button '(0))
           (list (op-button "+" +)
                  (op-button "-" -)
                  (op-button "*" *)
                  (op-button "/" /)
                  (op-button "=" (lambda (tot new) new))))

(make-calculator)
(make-calculator)
When to use State

Use state and `set!` when

• a function needs to remember something about previous calls, and

• you have no control over the callers
When NOT to use State

An unacceptable use of set!:

```
(define REV empty)
(define (reverse-list l)
  (cond
    [(empty? l) REV]
    [(cons? l)
      (begin
        (set! REV (cons (first l) REV))
        (reverse-list (rest l)))]))
(reverse-list '(1 2 3 4 5))
```

- Recursive calls build on earlier results, but we control all of the recursive calls
- It produces the wrong result when you call it a second time