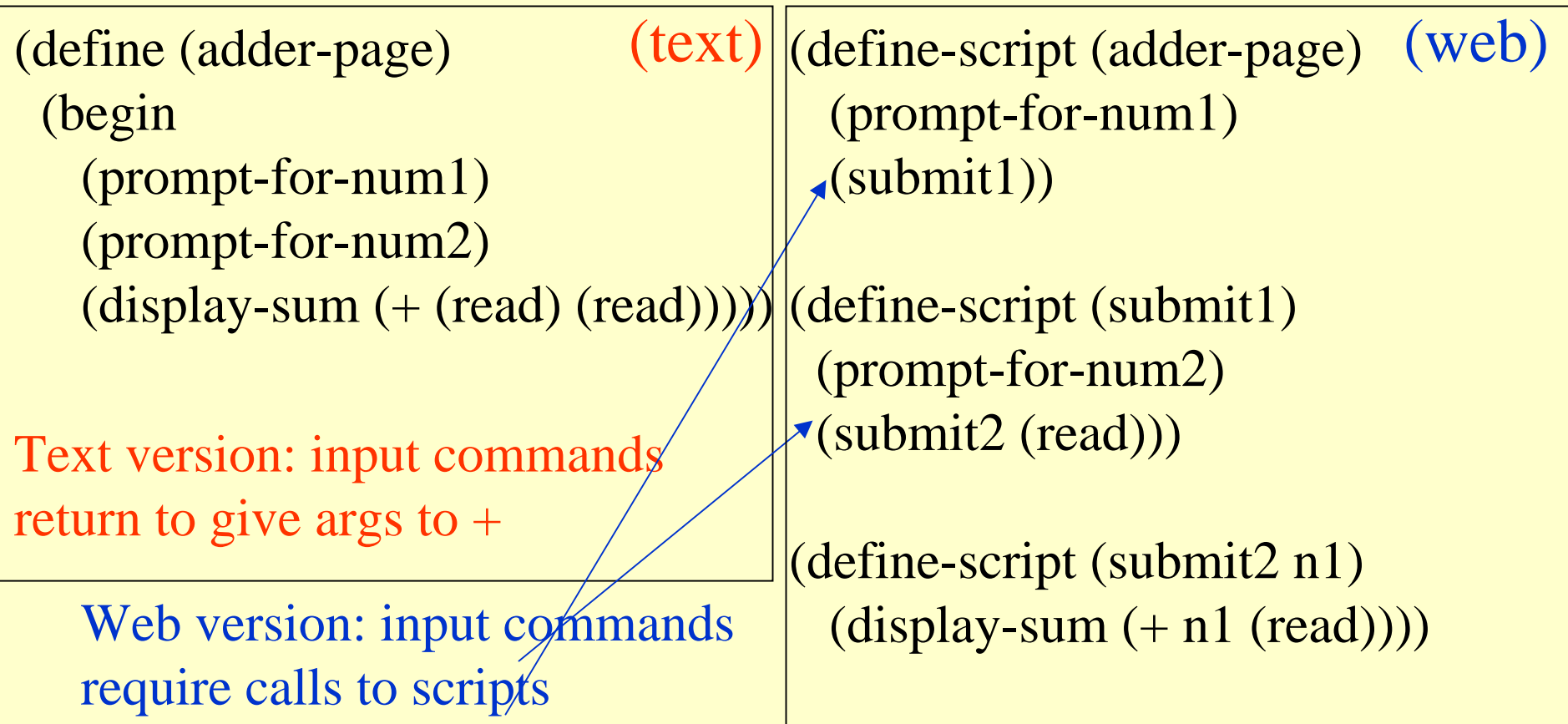


CS2135

Lectures on Script Position  
(aka: the “web compiler”)

# Recap: Web (CGI) scripts

- In the CGI protocol, there is no way to send the result of a script back to the page that invoked it. Each script must generate a new page to continue the computation.



# Recap: Web (CGI) scripts

- To use CGI, programmers are forced to structure their code (scripts) in a particular way [functions that handle input can't return to a pending operation]

<pre>(define (adder-page)   (begin     (prompt-for-num1)     (prompt-for-num2)     (printf "~a" (+ (read) (read))))))</pre> <p>Text version: input commands return to give args to +</p>	<p>(text)</p>	<pre>(define-script (adder-page)   (prompt-for-num1)   (submit1))  (define-script (submit1)   (prompt-for-num2)   (submit2 (read)))  (define-script (submit2 n1)   (printf "~a" (+ n1 (read))))</pre> <p>(web)</p>
--	---------------	--

Web version: input commands  
require calls to scripts

# Why does the difference in structure matter?

- What if a programmer needed two versions of the program (one for the web, one for another interface) – can't reuse the code ...
- ... and would have to maintain two versions.
- Better programming tools exist for testing and debugging non-web versions.
- Difference complicates the programmers' job

# How does our CS training tell us to address this problem?

Implement a program to convert programs to the web!

- Programmers write the text-I/O version
- Conversion program produces the web scripts
- When program changes, edit the text version and re-generate the scripts

This set of lectures will teach you how the conversion to web-format works

# Why are We Covering This?

- Web conversion illustrates the kinds of programs we can write to process other programs – programs that process other programs is one of the themes of this course.
- Shows you how programming languages and programming technology can impact a real-world application area such as the web
- The “web conversion” method actually applies to many other language issues

On to the conversion details ...

# Terminology: Script Position

A function call is in **script-position** in an expression if the value returned from the call is the value of the entire expression. Examples:

- $(f\ 4)$  is not in script position in  $(+ (f\ 4)\ 3)$   
[because the  $+$  needs the value of  $(f\ 4)$ ]
- The call to  $f$  in  $(f (+\ 4\ 3))$  is in script position  
[because the result from  $f$  is the result of the whole expression]



# Terminology: Script Position

A function call is in **script-position** in an expression if the value returned from the call is the value of the entire expression. Examples:

- `(g 6)` is not in script position in `(f (g 6))`
- the call to `f` is in script position in `(f (g 6))`
- `(f 6)` is in script position in

```
(cond [(= n 0) 1]
      [else (f 6)])
```

because `(f 6)` is the value of the `cond` (if else case is taken).

# Try it: which calls are in script position?

- `(* (f 3) (g 5))`
- `(cond [(foo 4) (bar 6)]  
[else (baz 7)])`
- `(f (cond [(= x 0) (h 2)]  
[else 9]))`

# Try it: which calls are in script position?

- `(* (f 3) (g 5))`

Call to `*` is not because only care about user-defined functions

- `(cond [(foo 4) (bar 6)]  
[else (baz 7)])`

Call to `foo` is not because result of `cond` is result of answers

- `(f (cond [(= x 0) (h 2)]  
[else 9]))`

Call to `h` is not because `f` is waiting for the result of the `cond`

# Try it: which calls are in script position?

```
(define (adder-page)
  (begin
    (prompt-for-num1)
    (prompt-for-num2)
    (display-sum (+ (read) (read))))))
```

(text)

```
(define-script (adder-page) (web)
```

```
  (prompt-for-num1)
  (submit1))
```

```
(define-script (submit1)
  (prompt-for-num2)
  (submit2 (read)))
```

```
(define-script (submit2 n1)
  (display-sum (+ n1 (read))))
```

# Try it: which calls are in script position?

```
(define (adder-page)
  (begin
    (prompt-for-num1)
    (prompt-for-num2)
    (display-sum (+ (read) (read)))))
```

(text)

```
(define-script (adder-page) (web)
```

```
  (prompt-for-num1)
  (submit1))
```

```
(define-script (submit1)
  (prompt-for-num2)
  (submit2 (read)))
```

```
(define-script (submit2 n1)
  (display-sum (+ n1 (read))))
```

Not surprisingly, the calls in the web version are all in script position (unlike reads)

# What's Important About Script Position?

- Calls that are in script position have the key characteristic of calls to web scripts: no pending computation is waiting for the result of the call
- Calls in script position can therefore be turned directly into invocations of web scripts

Task: We need to move all calls to user-defined functions that are **not** in script position into script position

# Moving Calls to Script Position

We can move calls to script position by following a simple sequence of steps. (We'll use the name "script" in functions that should be called only in script position)

```
(define (fscript x) (+ x 3))
```

```
(* (fscript 5) 2)           [returns 16]
```

# Moving Calls to Script Position

We can move calls to script position by following a simple sequence of steps. (We'll use the name "script" in functions that should be called only in script position)

```
(define (fscript x) (+ x 3))
```

```
(* (fscript 5) 2) [returns 16]
```

Call is not in script position – must move

Where would this call need to go to be in script position?

Must be at the front (nothing waiting for its result)



# Moving Calls to Script Position

We can move calls to script position by following a simple sequence of steps. (We'll use the name "script" in functions that should be called only in script position)

```
(define (fscript x) (+ x 3))
```

```
(* (fscript 5) 2) [returns 16]
```

---

```
(define (fscript x) (+ x 3))
```

```
(fscript 5) (* 2) [returns 16]
```

But this leaves a hole in the original expression!

Must be at the front (nothing waiting for its result)

# Moving Calls to Script Position

We can move calls to script position by following a simple sequence of steps. (We'll use the name "script" in functions that should be called only in script position)

```
(define (fscript x) (+ x 3))
```

```
(* (fscript 5) 2) [returns 16]
```

---

```
(define (fscript x) (+ x 3))
```

```
(fscript 5) (* hole 2) [returns 16]
```

But this leaves a hole in the original expression!

... so introduce a variable for the "hole"

Must be at the front (nothing waiting for its result)



# Moving Calls to Script Position

Now we need to create one expression out of the call to `fscript` and the lambda that will result in 16.

```
(define (fscript x) (+ x 3))
```

```
(fscript 5) (lambda (hole) (* hole 2))    [should return 16]
```

---

What is the relationship between `(fscript 5)` and `hole`?

- The result from `(fscript 5)` needs to eventually plug the hole (since we created the hole when we moved `(fscript 5)` to the front).
- But, we need to leave `fscript` at the front of the expression (so it stays in script position)

# Moving Calls to Script Position

Now we need to create one expression out of the call to `fscript` and the lambda that will result in 16.

```
(define (fscript x) (+ x 3))
```

```
(fscript 5) (lambda (hole) (* hole 2))    [should return 16]
```

---

We could edit the contract on `fscript` so that it takes a function (**action**) telling it what to do with its result ...

```
(define (fscript x action) (action (+ x 3)))
```

```
(fscript 5) (lambda (hole) (* hole 2))    [should return 16]
```

Now, just pass the lambda expression as the action to `fscript`!

# Moving Calls to Script Position

Now we need to create one expression out of the call to `fscript` and the lambda that will result in 16.

```
(define (fscript x) (+ x 3))
```

```
(fscript 5) (lambda (hole) (* hole 2))    [should return 16]
```

---

We could edit the contract on `fscript` so that it takes a function (**action**) telling it what to do with its result ...

```
(define (fscript x action) (action (+ x 3)))
```

```
(fscript 5 (lambda (hole) (* hole 2)))    [should return 16]
```

Now, just pass the lambda expression as the action to `fscript`!

# Recap: Result of First Example

We started with:

```
(define (fscript x) (+ x 3))  
(* (fscript 5) 2)           [returns 16]
```

And ended up with:

```
(define (fscript x action) (action (+ x 3)))  
(fscript 5 (lambda (hole) (* hole 2)))
```

Does the new expression also return 16? **Yes!**

## One Minor Edit ...

To avoid confusing the original fscript with the converted one, we'll adopt the convention of adding `/web` to the name of the function when we add the action parameter:

```
(define (fscript/web x action) (action (+ x 3)))  
(fscript/web 5 (lambda (hole) (* hole 2)))
```

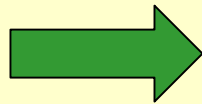
Note: the parameter name “action” is suggestive, because the function you pass to action behaves like the script that you specify in the action tag in an html form ...



# Summary: How to Convert a Function FooScript to FooScript/web

- Change the name and add a parameter called action
- Wherever FooScript returns a value, FooScript/Web should pass that value to action. If the body of FooScript is a cond, call action on each answer.

```
(define (myscript x)  
  (cond [(= x 4) 5]  
        [else 7]))
```



```
(define (myscript/web x action)  
  (cond [(= x 4) (action 5)]  
        [else (action 7)]))
```

# Summary: How to Move Calls into Script Position

- Find the script call that would happen first and move it to the front of the expression (or enclosing lambda)
- Replace the expression you moved with a variable (like hole – use different names each time you move a call within the same original expression though)
- Wrap a (lambda (hole) ...) around the original expression (minus the part you pulled out) and pass this lambda expression as an argument to the script call that you moved to the front.
- Repeat until all script calls are in script position

# Time to Try It! – Exercise 1

Move all calls to the scripts into script position (and modify the scripts as needed):

```
(define (fscript x y)
  (cond [(= x y) 6]
        [else (- y x)]))
```

```
(* (+ 4 5) (fscript 10 12))
```

```
(define (fscript\web x y action)
  (cond [(= x y) (action 6)]
        [else (action (- y x))]))
```

```
(fscript\web 10 12
  (lambda (hole)
    (* (+ 4 5) hole)))
```

# Time to Try It! – Exercise 2

```
(define (gscript x) (* x 6))
```

```
(define (hscript y) (+ y 5))
```

```
(+ 4 (gscript (hscript 3)))
```

```
(define (gscript/web x action)  
  (action (* x 6)))
```

```
(define (hscript/web y action)  
  (action (+ y 5)))
```

Move the call to `hscript` first because it executes first

```
(hscript/web 3 (lambda (box1)
```

```
  (+ 4 (gscript box1))))
```

# Time to Try It! – Exercise 2

```
(define (gscript x) (* x 6))
```

```
(define (hscript y) (+ y 5))
```

```
(+ 4 (gscript (hscript 3)))
```

```
(define (gscript/web x action)
```

```
(action (* x 6))
```

```
(define (hscript/web y action)
```

```
(action (+ y 5))
```

Now move call to gscript, but leave it inside the lambda

```
(hscript/web 3 (lambda (box1)
```

```
(+ 4 (gscript box1))))
```

# Time to Try It! – Exercise 2

```
(define (gscript x) (* x 6))
```

```
(define (hscript y) (+ y 5))
```

```
(+ 4 (gscript (hscript 3)))
```

```
(define (gscript/web x action)
```

```
(action (* x 6))
```

```
(define (hscript/web y action)
```

```
(action (+ y 5))
```

Now move call to gscript, but leave it inside the lambda

```
(hscript/web 3 (lambda (box1)
```

```
(gscript box1
```

```
(lambda (box2)
```

```
(+ 4 box2))))))
```

The call to gscript must remain in the lambda so that hscript runs first

# Time to Try It! – Exercise 3

```
(output "The answer is "
```

```
  (+ (prompt-read-script "First number: ")
```

```
    (prompt-read-script "Second number: ")))
```

Which call goes to the outside first?

```
(prompt-read-script
```

```
  "First number: "
```

```
  (lambda (box1)
```

```
    (output "The answer is"
```

```
      (+ box1 (prompt-read-script "Second number: "))))))
```

Now continue with the second prompt call ...

# Time to Try It! – Exercise 3

(output "The answer is "

(+ (prompt-read-script "First number: ")

(prompt-read-script "Second number: ")))

Now continue with the second prompt call ...

(prompt-read-script

“First number: ”

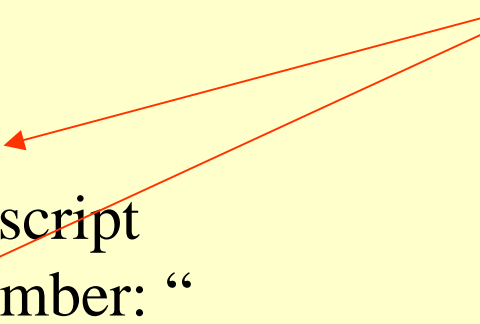
(lambda (box1)

(prompt-read-script

“Second number: “

(lambda (box2) (output (+ box1 box2))))))

If you name these lambdas,  
you get code that looks like  
the web version of the  
project!





# Time to Try It! – Exercise 4

```
(define (countdown n)
  (cond [(zero? n) (output "Liftoff!")]
        [else (begin
                  (prompt-read-script (format "t - ~a and counting" n))
                  (countdown (- n 1))))])

(countdown (prompt-read-script "Time left on launch pad: "))
```

---

Think of this as a program for NASA: as countdown progresses, the program produces a progress report and expects the user to give some input to continue the countdown (the program ignores the content of the input though).

First, convert the expression that calls countdown

# Time to Try It! – Exercise 4

```
(define (countdown n)
  (cond [(zero? n) (output "Liftoff!")]
        [else (begin
                  (prompt-read-script (format "t - ~a and counting" n))
                  (countdown (- n 1))))]))

(countdown (prompt-read-script "Time left on launch pad: "))
```

---

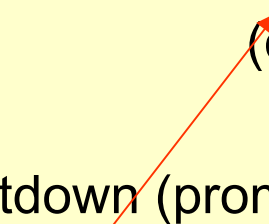
```
(prompt-read/web "Time left on launch pad: "
                 (lambda (box) (countdown box)))
```

**First, convert the expression that calls countdown**

# Time to Try It! – Exercise 4

```
(define (countdown n)
  (cond [(zero? n) (output "Liftoff!")]
        [else (begin
                  (prompt-read-script (format "t - ~a and counting" n))
                  (countdown (- n 1)))]))

(countdown (prompt-read-script "Time left on launch pad: "))
```



Now, move the call to `prompt-read-script` inside `countdown`

Where should this call move to? [think about it]

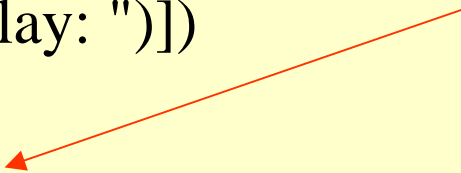
It should stay inside the `else`: remember, `cond` answers are in script position and we shouldn't call the script if `n` is zero



# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

Here, the  
script is  
recursive



```
(output "Total delay: " (count-delay-script))
```

---

This program asks a user to keep entering numbers at a prompt.  
When the user enters zero, the program prints the sum and stops.

Once again, convert the output expression first ...



```
(count-delay/web (lambda (box)
                  (output "Total delay: " box)))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Now, rename `count-delay-script` and add `action` parameter

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Next, send cond answers to action ...

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Next, send cond answers to action ...

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) (action 0)]
          [else (action (+ new-delay (count-delay-script)))])))
```



# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Now, look for calls to scripts to move

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) (action 0)]
          [else (action (+ new-delay (count-delay-script)))])))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Now, look for calls to scripts to move

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) (action 0)]
          [else (action (+ new-delay (count-delay-script)))])))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

The `prompt-read-script` call happens first, so move it first

```
(define (count-delay/web action)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) (action 0)]
          [else (action (+ new-delay (count-delay-script)))])))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

The `prompt-read-script` call happens first, so move it first

```
(define (count-delay/web action)
  (prompt-read/web "Delay: ")
  (lambda (box1)
    (let ([new-delay box1])
      (cond [(= 0 new-delay) (action 0)]
            [else (action (+ new-delay (count-delay-script)))]))))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Now move the `count-delay-script` call (and rename to `count-delay/web`)

```
(define (count-delay/web action)
  (prompt-read/web "Delay: ")
  (lambda (box1)
    (let ([new-delay box1])
      (cond [(= 0 new-delay) (action 0)]
            [else (action (+ new-delay (count-delay-script)))]))))
```

# Time to Try It! – Exercise 5

```
(define (count-delay-script)
  (let ([new-delay (prompt-read-script "Delay: ")])
    (cond [(= 0 new-delay) 0]
          [else (+ new-delay (count-delay-script))])))
```

---

Now move the `count-delay-script` call (and rename to `count-delay/web`)

```
(define (count-delay/web action)
```

```
(prompt-read/web "Delay: "
```

```
(lambda (box1)
```

```
(let ([new-delay box1])
```

```
(cond [(= 0 new-delay) (action 0)]
```

```
[else (count-delay/web
```

```
(lambda (box2) (action (+ new-delay box2)))]))
```

Moral: recursive scripts  
aren't harder to convert –  
follow same rules

# Script Position: Recap

- Now done several examples, including recursive scripts
- After conversion, all calls to scripts are in “script position”, so can easily produce versions that would run on a webserver
- In 2002, student used this to write a web-based version of Mastermind – used the conversion to get the looping right.

# What Else is Script Position Useful?

Script position also helps program optimization (compilers).

Consider sum (of list of numbers):

```
(define (sum alon)
  (cond [(empty? alon) 0]
        [(cons? alon) (+ (first alon) (sum (rest alon)))]))
```

*not in script position* (with arrow pointing to the recursive call)

---

```
(sum (list 1 3 5 7))
(+ 1 (sum (list 3 5 7)))
(+ 1 (+ 3 (sum (list 5 7))))
(+ 1 (+ 3 (+ 5 (sum (list 7)))))
(+ 1 (+ 3 (+ 5 (+ 7 (sum empty)))))
(+ 1 (+ 3 (+ 5 (+ 7 0))))
(+ 1 (+ 3 (+ 5 7)))
(+ 1 (+ 3 12))
(+ 1 15)
16
```

Example of run of  
sum and main steps in  
execution

Notice the shape of  
the computation –  
grows out then shrinks



# What Else is Script Position Useful?

Here's a version of sum with recursive calls in script position

```
(define (sum2 alon) (sum-help alon 0))
(define (sum-help alon total)
  (cond [(empty? alon) total]
        [(cons? alon) (sum-help (rest alon) (+ (first alon) total))]))
```

---

```
(sum2 (list 1 3 5 7))
(sum-help (list 1 3 5 7) 0)
(sum-help (list 3 5 7) 1)
(sum-help (list 5 7) 4)
(sum-help (list 7) 9)
(sum-help empty 16)
16
```

Notice the shape of the computation now – there are no pending calls, so the expression at each step has the same size

This version runs in less space than the original (won't run out of room on the stack, whereas original sum might)

# Summary

- Script Position (really called “tail position”) is a useful concept in programming languages
- Many compilers will convert certain calls to tail position as an optimization (so you can write code not in script position and still gain benefits of it)
- So why don't web servers do that? Stay tuned ...

For now, you should be able to convert calls to script position, as we did in these exercises