

CSE 1729: Principles of Programming

Lecture 8: Short Circuit Evaluation, Typing and Lambda Expressions



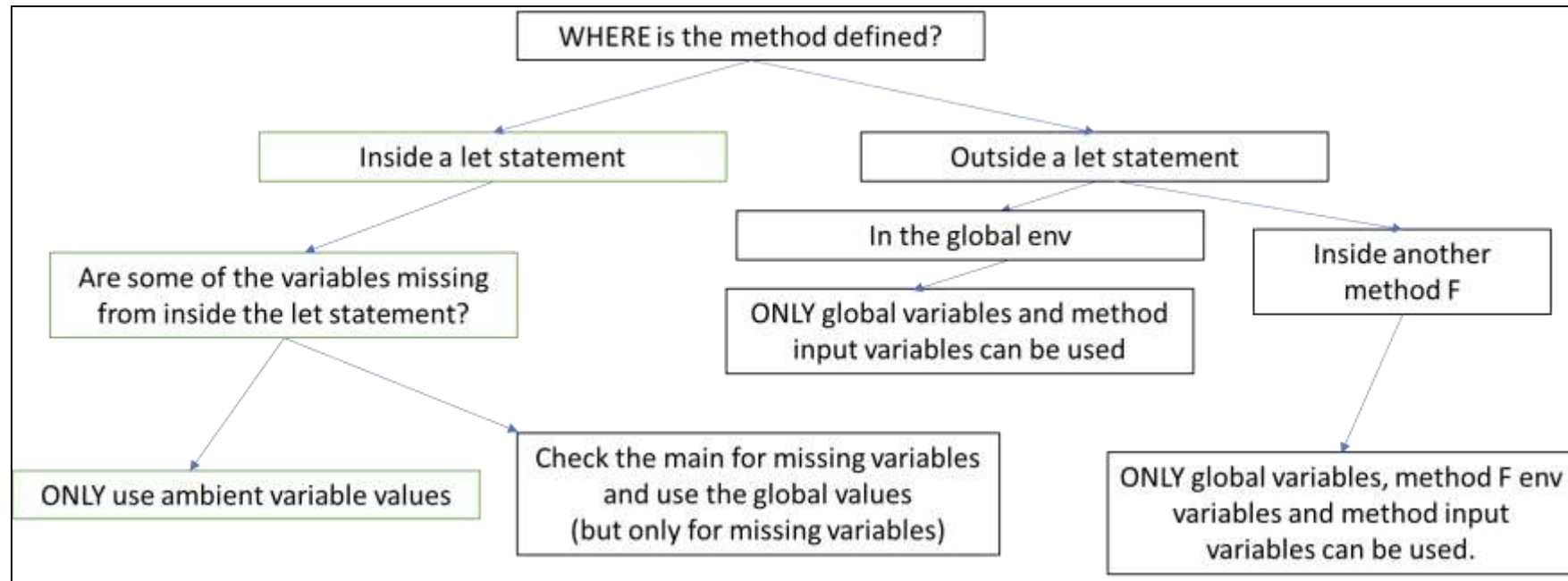
Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

Last time in CSE 1729...

1. We developed a guide to determine which environment we check for variable values:



Last time on the hit show CSE 1729...

2. We went over four “let” statement examples:

A method defined in the global environment and used inside of a let statement:

```
1 (define a 3)
2 (define constant 5)
3
4 (define (addConstant x)
5   (+ constant x)
6 )
7
8 (let ((constant 7))
9   (addConstant a))
```

A method defined in the let statement

```
1 (define a 0)
2 (define constant 0)
3 (let ((constant 7)
4     (a 3))
5   (define (addConstant x) (+ constant x))
6   (addConstant a)
7 )
```

Last time on the hit show CSE 1729...

2. We went over four “let” statement examples:

A method defined inside of a let statement then called inside of another let statement.

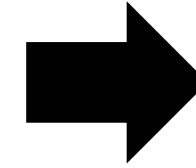
```
1 (let ((constant 7))
2   (define (addConstant x) (+ x constant))
3   (let ((constant 5))
4     (addConstant 3)))
5
```

A method defined inside another method, then called within a let statement in the method.

```
(define (f x)
  (define (g y)
    (+ x y))
  (let ((x 5))
    (g 11)))
> (f 6)
```

One more example to confuse you...

```
1 (define a 1)
2 (let ((x 3))
3   (define (doubleNumber x) (+ x x)))
4   (doubleNumber a))
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
2
>
```

GLOBAL Environment

a = 3

Ambient Environment

x = 3

Method Environment

x = 1

Remember when we define methods with input parameters like “x” in line 3:
“x” is just a placeholder variable.

REMINDER ABOUT THE LIFE AND TIMES OF AN ENVIRONMENT...

- Environments contain bindings of variables to values.
 - The define command destructively adds a binding to an environment.
 - There are two ways that new environments are created:
 - During function evaluation.
 - During let evaluation.
- (Actually, these are the same internal process)
- These new environments **always** inherit all bindings from the environment from which they were created: however, the bindings of arguments shadow existing bindings.

WITH LEXICAL SCOPE...

FUNCTIONS BEHAVE LIKE FUNCTIONS!

- “Free” variables in the body of a scheme function are assigned values from the environment in which the function was *defined*.
- This makes reasoning about their values easy, they are always drawn from the same environment!
- When definition environments never change, we are using *functional programming*.
- There are some cool things you can do by fiddling with definitional environments *after a function has been defined*. Part III of the course.

Example: Determining Divisibility

- Problem: Given number n . Find the largest number k that divides n without remainder. The maximum value of k can be $n - 1$.

First think of simple cases:

$$n = 2 \text{ so } k = 1$$

$$n = 70 \text{ so } k = 35$$

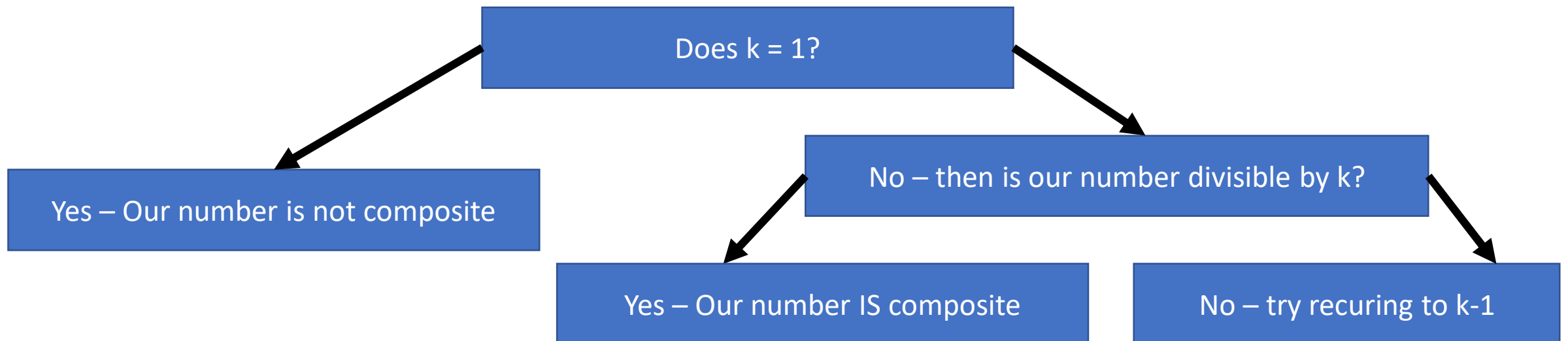
- A number n is prime if $k = 1$.
- A number n is composite if $k \neq 1$ (i.e., the number is not prime).

Example Python Code

```
1  def determineIfComposite(n):
2      k = 1
3      for i in range(2, n): #Go through all values up to n-1
4          #Check if divisible and larger than current solution
5          if n % i == 0 and i>k:
6              k = i
7      #Check if prime or not
8      if k == 1:
9          return False
10     else:
11         return True
```

Checking for Composite In Scheme

```
(define (divides a b) (= (modulo b a) 0))  
(define (smooth n k)  
  (if (< k 2)  
      #f  
      (if (divides k n)  
          #t  
          (smooth n (- k 1))))))
```



Can we try writing this using AND/OR statements?

- First question. What are AND / OR operations?
- Note in the table below 0 is equivalent to false, 1 is equivalent to true.

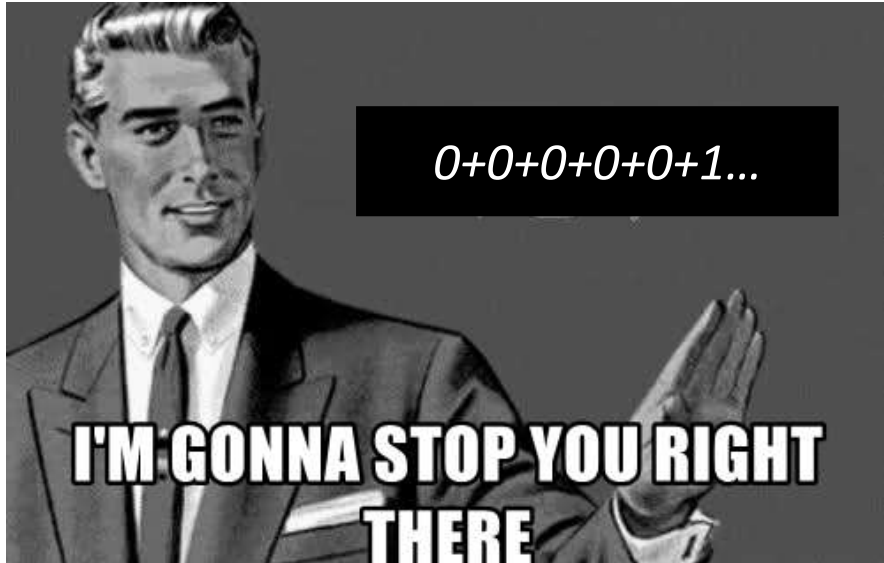
AND Truth Table

Inputs		Output
A	B	$Y = A.B$
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

Inputs		Output
A	B	$Y = A+B$
0	0	0
0	1	1
1	0	1
1	1	1

Some Important Properties of AND/OR



- **False** AND True AND True AND True...= False
- This is the same thing as saying $0*1*1*1...=0$. We don't need to care about the other numbers in the sequence!
- We can think about the same thing for OR
- **True** OR False OR False OR False...=True
- The is the same as $1+0+0+0...=1$. We don't need to care about the other numbers in the sequence!

Short Circuit Evaluation

- When using AND/OR operations Scheme will automatically stop evaluating and return when it sees one of the two “short cuts” that we mentioned above.
- Don’t take my word for it. Try it!

```
1 (define (recur x)
2   (display x) (newline)
3   (or #t (recur x))
4   )
5
6 (recur 5)
```

How many times should this code recur? An infinite number of times! So we should see 5 being printed repeatedly...

```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
5
#t
```

Back to the Composite Problem

Can we write this using AND/OR statements? Hint: yes

```
(define (divides a b) (= (modulo b a) 0))  
(define (smooth n k)  
  (if (< k 2)  
      #f  
      (if (divides k n)  
          #t  
          (smooth n (- k 1))))))
```

How would we write the conditions for determining if a number is composite?

Remember if $k \% n == 0$ the number is composite
(when $k \neq 1$)

Start with k should not be equal to 1

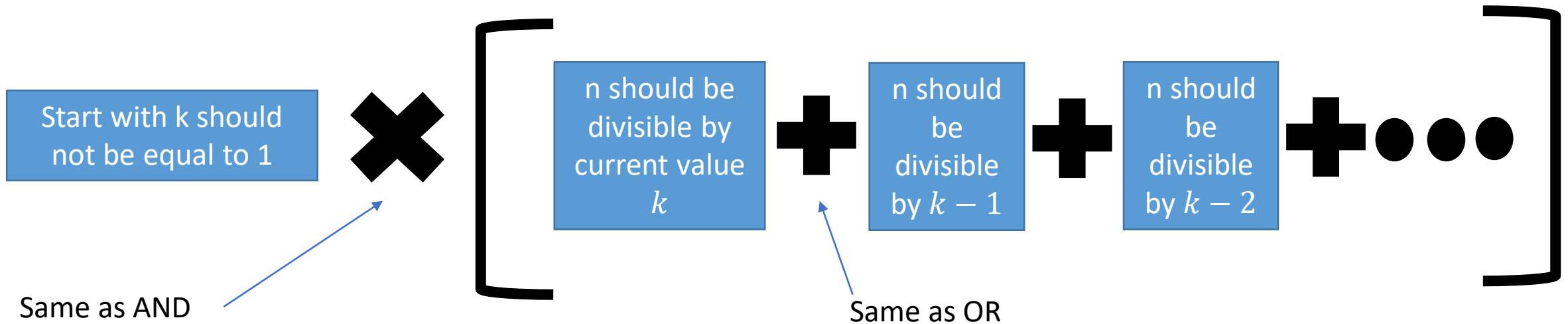
AND

it should be divisible by current value k

OR it should be divisible by $k - 1$

OR it should be divisible by $k - 2$

Determining Composite Using Short Circuit Evaluation



```
(define (divides a b) (= (modulo b a) 0))
(define (smooth n k)
  (and (>= k 2)
        (or (divides k n)
              (smooth n (- k 1)))))

(define (composite n) (smooth n (- n 1)))
```


Like short circuit computation, you only need to read the FIRST correct answer to get the question right...



Why is short circuit evaluation good?

• A: It lets you shorten computation time

• B: Trafalgar Square

• C: Tower of London

• D: Buckingham Palace

More on Functions in Scheme

IN SCHEME, FUNCTIONS ARE *FIRST CLASS OBJECTS*.

- Functions can be passed as arguments: Consider the following definition:

A function A positive integer

```
(define (sum f n)
  (if (= n 0)
      (f 0)
      (+ (f n) (sum f (- n 1)))))
```

This computes the sum: $f(0) + f(1) + \dots + f(n)$

Both f and n are passed as arguments.

THEN...

- To compute the sum of the first n squares: $0^2 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2$

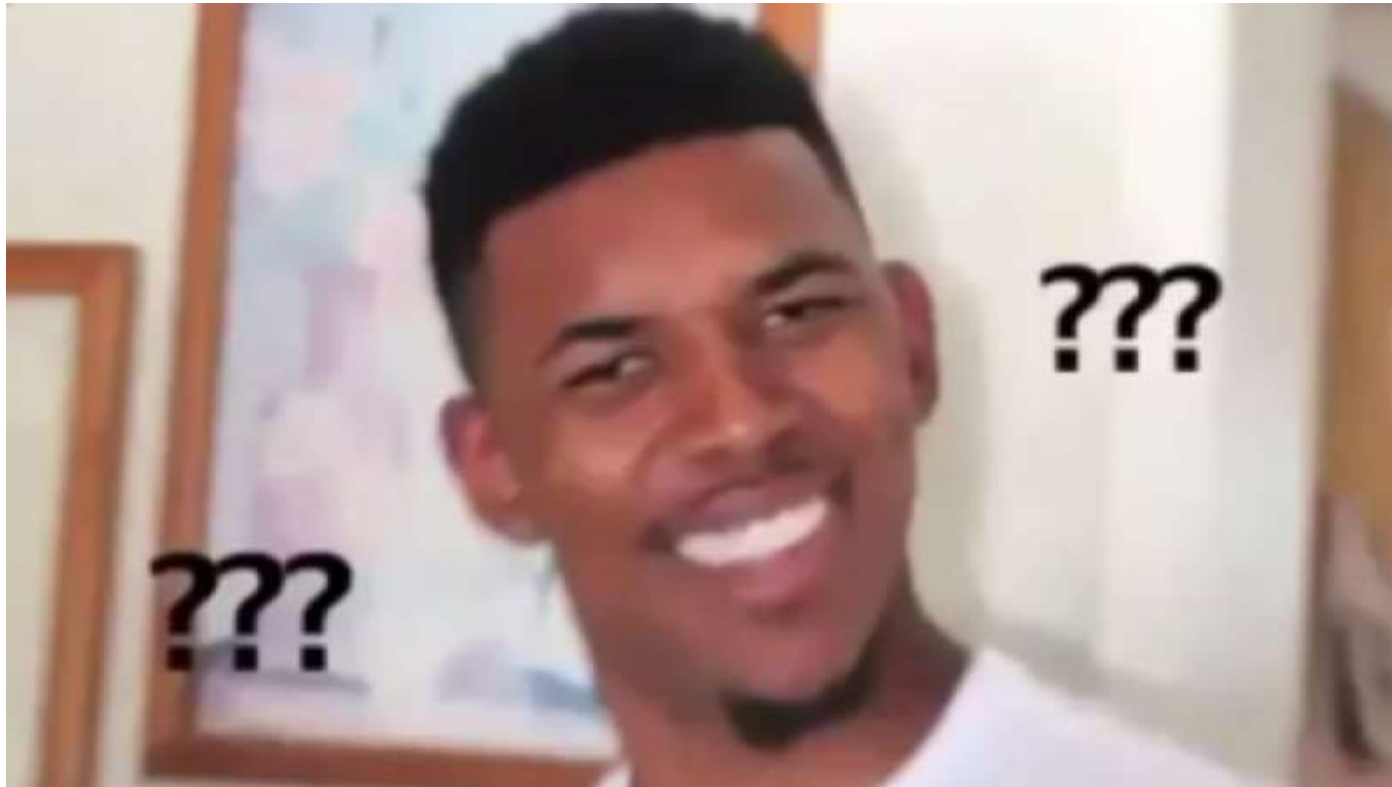
```
> (define (square x) (* x x))  
> (sum square 5)  
55
```

← Square(5)+Square(4)+Square(3)+Square(2)+Square(2)+Square(1)+Square(0)

- To compute the sum of the first n cubes: $0^3 + 1^3 + 2^3 + 3^3 + 4^3 + 5^3$

```
> (define (third x) (* x x x))  
> (sum third 5)  
225
```

Question: *Given a method definition in Scheme, how would you know if the input parameters are supposed to be values or functions?*



Strongly typed vs Weakly typed

- Accordingly to Wikipedia “Strongly” and “Weakly” typed languages are colloquial definitions. However, we can still think about such ideas in terms of error handling and exchanging code?
- Let’s look at the language C# in which all variables must have a declared type. Some things to note:
 - Methods in C# have to be declared public/private (and sometimes static). You don’t need to worry about those keywords in this class.

What you should pay attention to:

Inputs to the method must have a pre-defined type (e.g. double means the number has 64 bits of precision to represent a decimal number).

The Square Function in C#

Don't worry about static or public keywords for now

Indicates the return type

Name of the function

```
26 public static double square(double x)
27 {
28     return x * x;
29 }
```

Indicates the type for the method input parameter

Actual variable name for the input

Writing the Sum Function in C#

```
14 //Sum takes in number of times it should be called and
15 //SomeMethod with double as input and double as output.
16 1 reference
17 public static double sum(int n, Func<double, double> SomeMethod)
18 {
19     double currentSum = 0; //Inititalize the current sum to 0
20     for(int i = 0; i < (n+1); i++)
21     {
22         currentSum = currentSum + SomeMethod(i);
23     }
24     return currentSum;
25 }
26 1 reference
27 public static double square(double x)
28 {
29     return x * x;
30 }
```


Comparing C# and Scheme

Scheme

```
1 (define (sum f n)
2   (if (= n 0)
3       (f 0)
4       (+ (f n) (sum f (- n 1)))))
5
6 (define (square x) (* x x))
```

You must declare the *variable type*...hence C# is more strongly typed than Scheme in this respect.

C#

```
//Sum takes in number of times it should be called and
//SomeMethod with double as input and double as output.
1 reference
public static double sum(int n, Func<double, double> SomeMethod)
{
    double currentSum = 0; //Initialize the current sum to 0
    for(int i = 0; i < (n+1); i++)
    {
        currentSum = currentSum + SomeMethod(i);
    }
    return currentSum;
}

1 reference
public static double square(double x)
{
    return x * x;
}
```

What difference do you notice? (beside the for loop)

What is the advantage of strongly typed?

What is the advantage of weakly typed?



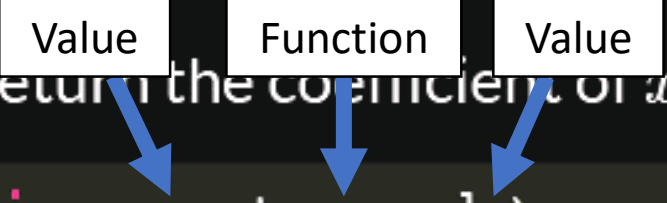
Now that you know these concepts you can become a coding wizard.

- Strongly typed means there may be less chance you will have a compilation error because it doesn't let you mix up variables. Can read other people's code much faster because you don't have to infer variable types.
- Weakly typed means you can write code faster (because you don't have to constantly be declaring variable types). Code may be more flexible because you don't have to write multiple definitions for different variables type.
- Can you think of other strengths and weakness of each approach?

Now back to Scheme...

ANOTHER EXAMPLE. A TOOL FOR PARTIAL POWER SERIES

- A power-series expander.
- term is a function that should return the coefficient of x^k .



```
(define (power-series x term k)
  (if (< k 0)
      0
      (+ (* (term k) (expt x k))
          (power-series x term (- k 1)))))
```

- Then (power-series x term k) should return:

$$term(0) + term(1)x + term(2)x^2 + \dots + term(k)x^k$$

UNDERSTANDING `sin-term`

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

- The coefficients of the even terms are always 0

$$\sin(x) \approx 1 \cdot x + 0 \cdot x^2 - \frac{1}{3!} \cdot x^3 + 0 \cdot x^4 + \frac{1}{5!} \cdot x^5 + 0 \cdot x^6 - \frac{1}{7!} \cdot x^7 + \dots$$

term(0) = 0 even

term(1) = 1

term(2) = 0 even

term(3) = - 1/3!

term(4) = 0 even

term(5) = + 1/5!

term(6) = 0 even

term(7) = - 1/7!

How to get the sign?
It alternates!

$$(-1)^{\frac{t-1}{2}}$$

Using the power series to approximate the Sin function

$$\sin(x) \approx 1 \cdot x + 0 \cdot x^2 - \frac{1}{3!} \cdot x^3 + 0 \cdot x^4 + \frac{1}{5!} \cdot x^5 + 0 \cdot x^6 - \frac{1}{7!} \cdot x^7 + \dots$$

GENERATING PARTIAL POWER SERIES

- sin and cos: Setting the stage

```
(define (fact n) (if (= n 0)
                    1
                    (* n (fact (- n 1)))))
(define (odd t) (= (modulo t 2) 1))
```

Just the
factorial
function

- The term definitions:

```
(define (sin-term t)
  (if (odd t) (/ (expt -1 (/ (- t 1) 2)) (fact t))
      0))
(define (cos-term t)
  (if (odd t) 0 (/ (expt -1 (/ t 2)) (fact t))))
```

Terms for
the sin
function

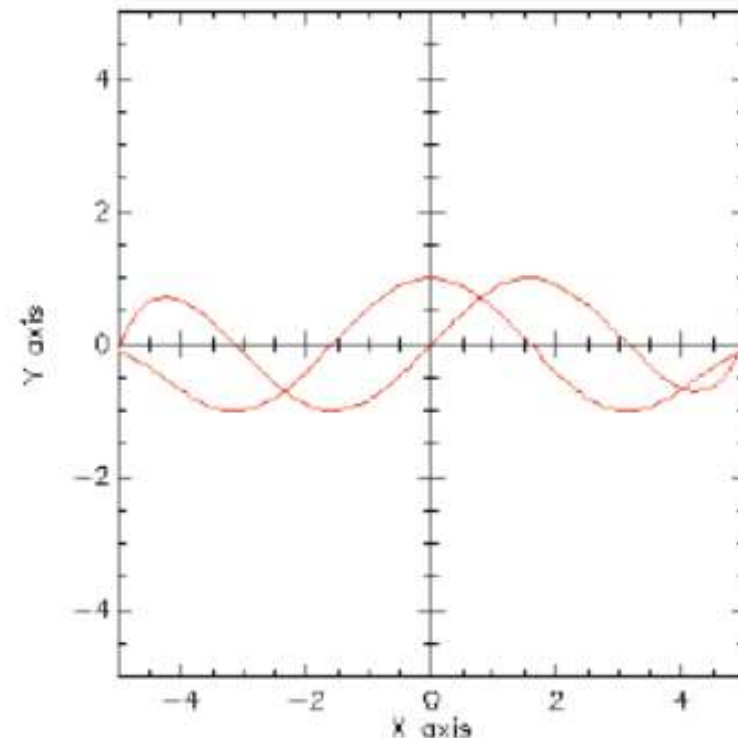
NOW THE POWER SERIES ARE EASY TO GENERATE

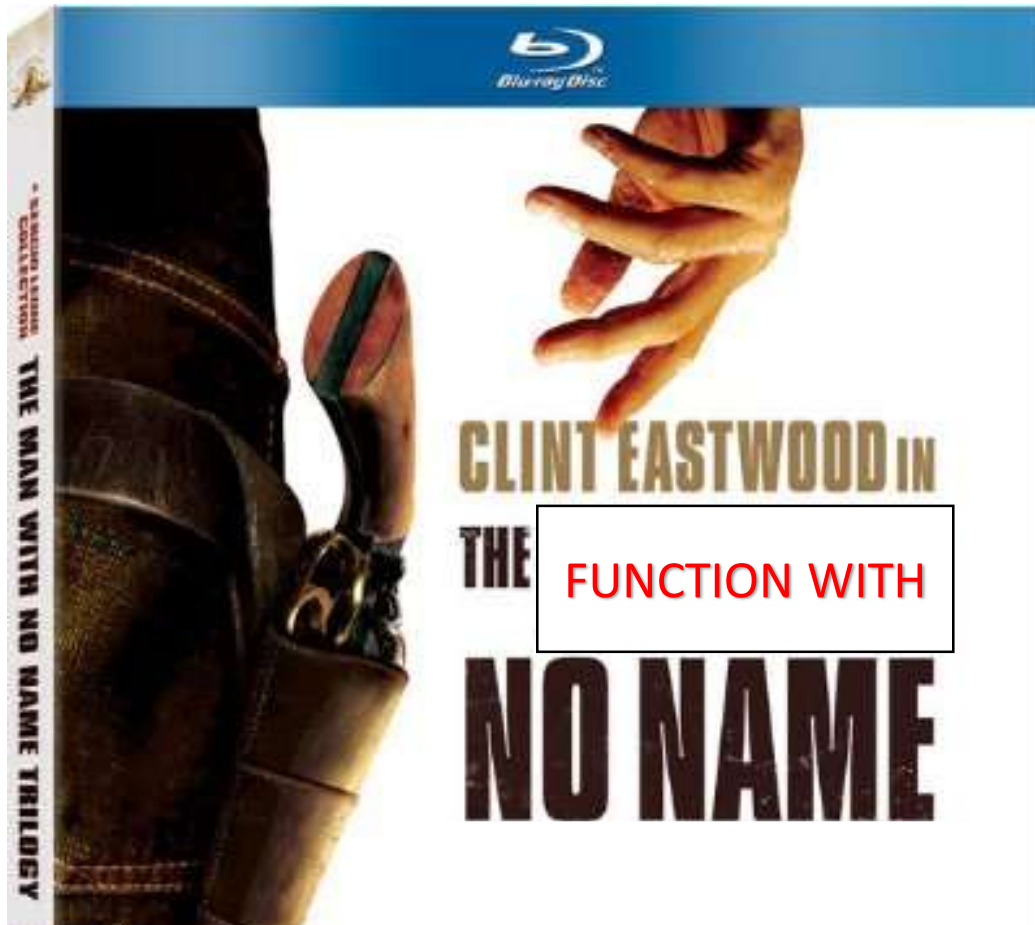
- Now we can define functions from the first 10 terms of each power series:

```
(define (sin10 x) (power-series x sin-term 10))  
(define (cos10 x) (power-series x cos-term 10))
```

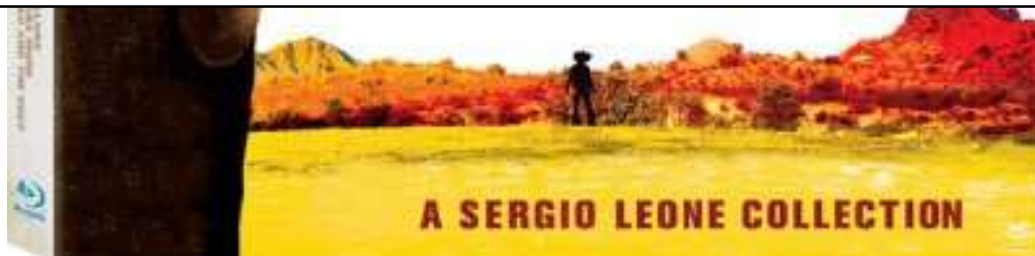
Then, for example,

```
(require plot)  
(plot (mix (line sin10)  
           (line cos10)))
```





Lambda Expressions: The Fastest Function in the West!!!



Functions with no name

- Scheme has a mechanism for defining functions without names:

argument body



```
(lambda (x) (* x x))
```

is the **function** that returns the square of its argument.

- If you wish to sum the values of the first n squares, instead of defining square first, you can directly pass the function:

```
> (sum (lambda (x) (* x x)) 10)  
385
```

DEFINE REVISITED

- If we enlarge our notion of value to include function values, we can simplify the definition of define as an operator that always binds a name to a value.

```
(define (square x) (* x x))
```

is the same as...

```
(define square (lambda (x) (* x x)))
```

let REVISITED

- We can express let using lambda and the standard application rule!

```
(let ((x1 <expr1>)
     ...
     (xk <exprk>))
  <let-expr>)
```

is the same as...

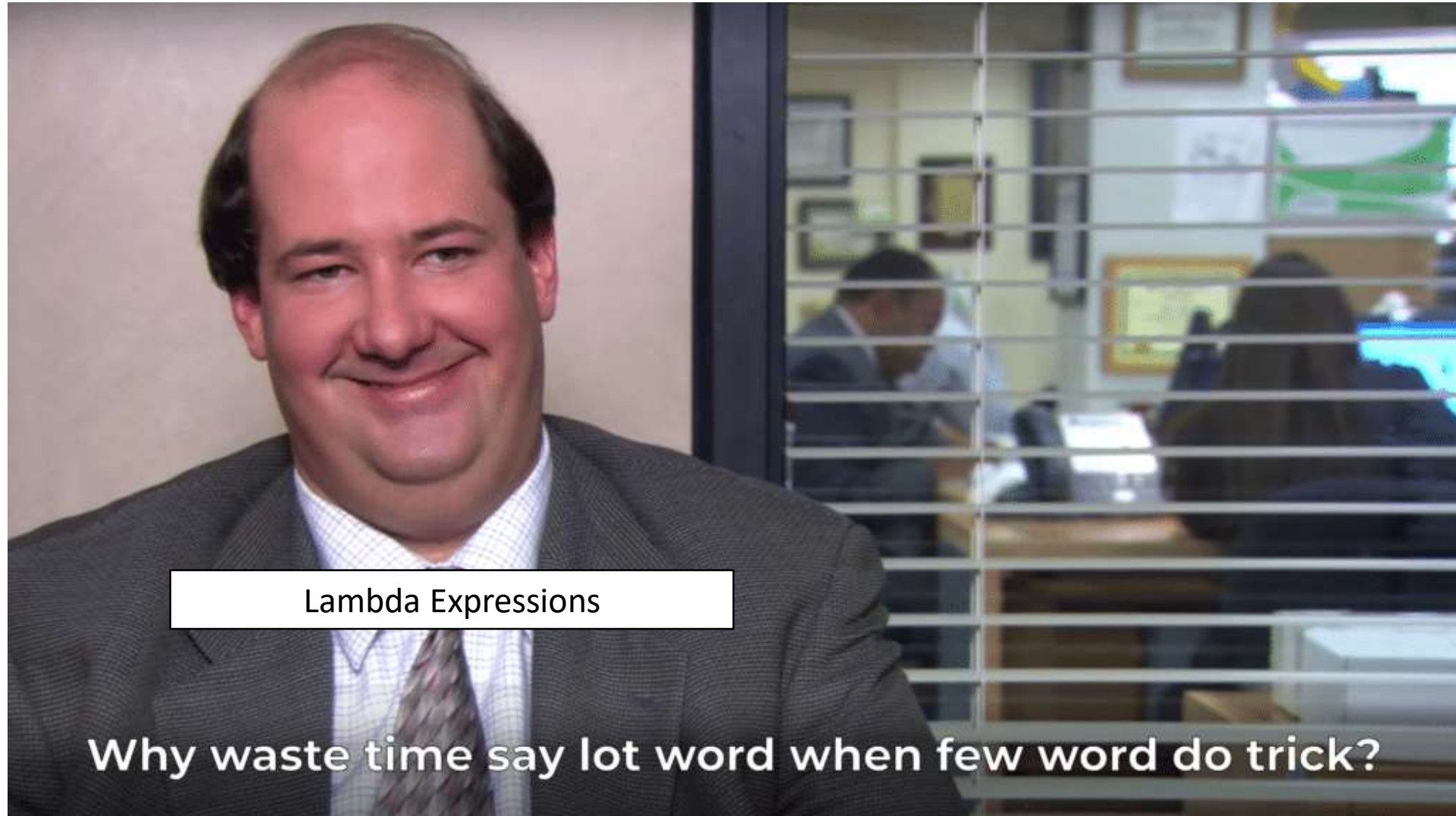
```
((lambda (x1 ... xk) <let-expr>)
  <expr1> ... <exprk>)
```

Example Equivalence between Let and Lambda

```
1  (let  ((x 5)
2         (y 4))
3         (+ x y))
4
5  ((lambda (x y) (+ x y)) 5 4)
6
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
9
9
>
```

What is the purpose of Lambda Expressions?



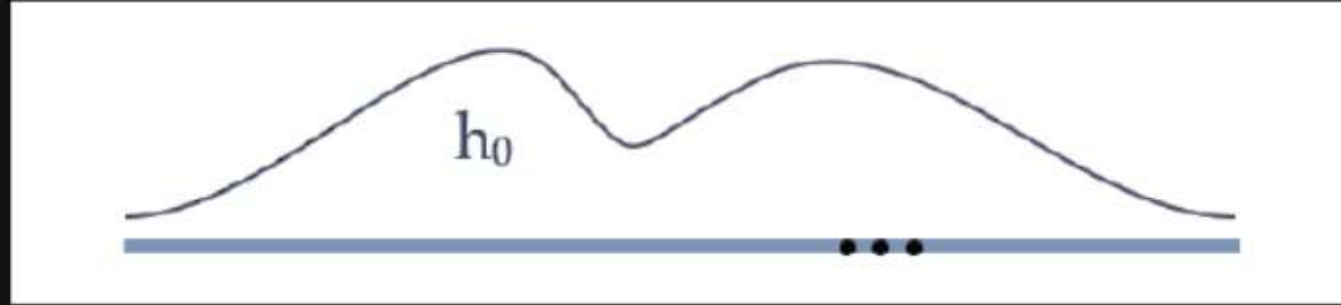
Lambda Expressions

Why waste time say lot word when few word do trick?

An Example in Python

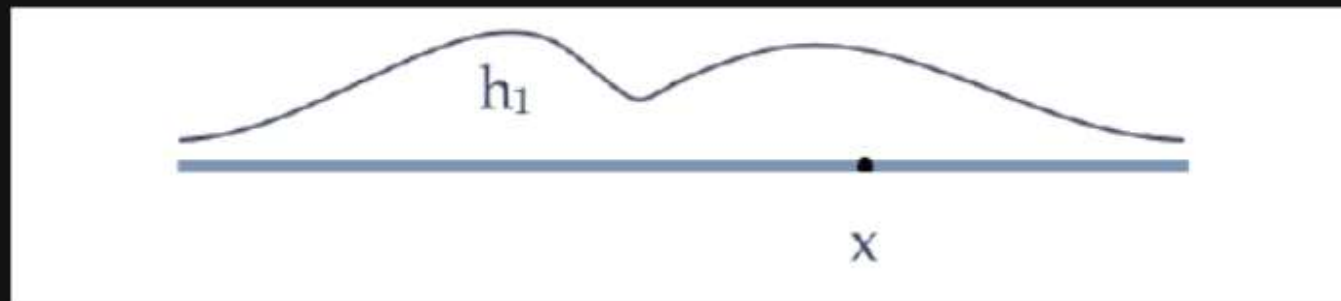
```
1  #Square function (with def)
2  def square(x):
3      squareResult = x*x
4      return squareResult
5  result = square(5) #Call the function
6
7  #Same calculation using the lambda expression
8  d = (lambda x: x * x)(5)
```

An Example Using Lambda Expressions: The Heat Equation



$$h_1(x) = \frac{h_0(x+dx) + h_0(x-dx)}{2}$$

The average of two
close points



RETURNING FUNCTIONS AS “VALUES”

- The lambda form provides an easy way to return a function as a value.

```
(define (heat-flow f dx)
  (lambda (x) (/ (+ (f (+ x dx))
                    (f (- x dx)))
                2)))
```

```
(define (heat-flow-evolve f dx t)
  (if (= t 0)
      f
      (heat-flow (heat-flow-evolve f dx (- t 1)) dx)))
```


Let's test it out with a simple function:

$f(x) = x^2$: Let me find the value using heat flow at time $t=5$.

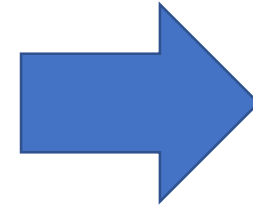
```
1 (define (fun x)
2   (* x x))
3
4 (define (heat-flow f dx)
5   (lambda (x) (/ (+ (f (+ x dx))
6                     (f (- x dx)))
7                 2))))
8
9 (define (heat-flow-evolve f dx t)
10  (if (= t 0)
11      f
12      (heat-flow (heat-flow-evolve f dx (- t 1)) dx)))
13
14 (heat-flow-evolve fun 0.2 5)
```

dx=0.2

t=5

What should the answer be? Close to 25 right?

```
1 (define (fun x)
2   (* x x))
3
4 (define (heat-flow f dx)
5   (lambda (x) (/ (+ (f (+ x dx))
6   (f (- x dx)))
7   2)))
8
9 (define (heat-flow-evolve f dx t)
10  (if (= t 0)
11      f
12      (heat-flow (heat-flow-evolve f dx (- t 1)) dx)))
13
14 (heat-flow-evolve fun 0.2 5)
```

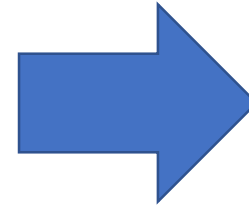


```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
#<procedure>
>
```



Heat flow example done right...

```
1 (define (fun x)
2   (* x x))
3
4 (define (heat-flow f dx)
5   (lambda (x) (/ (+ (f (+ x dx))
6   (f (- x dx)))
7   2)))
8
9 (define (heat-flow-evolve f dx t)
10  (if (= t 0)
11      f
12      (heat-flow (heat-flow-evolve f dx (- t 1)) dx)))
13
14 ;(heat-flow-evolve fun 0.2 5)
15 ((heat-flow-evolve fun 0.2 5) 5)
16
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
25.199999999999996
>
```

We are returned a function, not a value so we further need to evaluate the function with some value.

Figure Sources

- <https://i.ytimg.com/vi/JQ0JOfxUAgY/maxresdefault.jpg>
- <https://memegenerator.net/img/instances/35061693.jpg>
- https://metro.co.uk/wp-content/uploads/2019/01/SEI_45946208-eb15.jpg?quality=90&strip=all&zoom=1&resize=480%2C274
- <https://i.kym-cdn.com/entries/icons/original/000/018/489/nick-young-confused-face-300x256-nqlyaa.jpg>
- <https://o.aolcdn.com/images/dar/5845cadfec996e0372f/2fd687b948d4d7e871bdb3f1a1b4162b8ed9cddc/aHR0cDovL28uYW9sY2RuLmNvbS9oc3Mvc3RvcnFnZS9hZGFtLzc4YTYwNWNmYzk2ODJkYjgwMzg4MTZkZjM0OWRkOWUzL2dhbmRhbGYgbG90ciBtYWNib29rIGFwcGxlLmpwZw==>
- <https://m.media-amazon.com/images/I/51Ue8GNJ3aL.jpg>
- <https://m.media-amazon.com/images/M/MV5BMGRmNGFkMjctY2M2OC00NWU3LTljY2EtY2ViOWI4NjY1ODM1XkEyXkFqcGdeQXVyNDczMTAyNTg@. V1 .jpg>
- <https://sonder.com.au/wp-content/uploads/2020/10/quality-over-quantity-blog@2x-1024x576.png>
- <https://i.kym-cdn.com/photos/images/original/001/431/201/40f.png>