

Toyota robot can't slam dunk but shoots a mean 3-pointer

<https://www.bloomberg.com/news/articles/2019-04-03/amazon-schooled-on-ai-facial-technology-by-turing-award-winner>

“Researchers at Toyota Motor have developed a six-foot, 10-inch-tall robot that can make free throws and three-point shots on a basketball court.

The Cue 3 robot computes the basket’s location as a three-dimensional image using sensors on its torso, and adjusts motors inside its arm and knees to give the shot the correct angle and propulsion to go through the hoop.



Robots that can mimic human movements, even doing them better in some cases, could prove useful in various ways, including picking crops, making deliveries, and working in factories or warehouses. Said Stanford University’s Oussama Khatib, “What Toyota is doing here is really bringing the top capabilities in perception with the top capabilities in control to have robots perform something that is really challenging.”

Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was *defined*

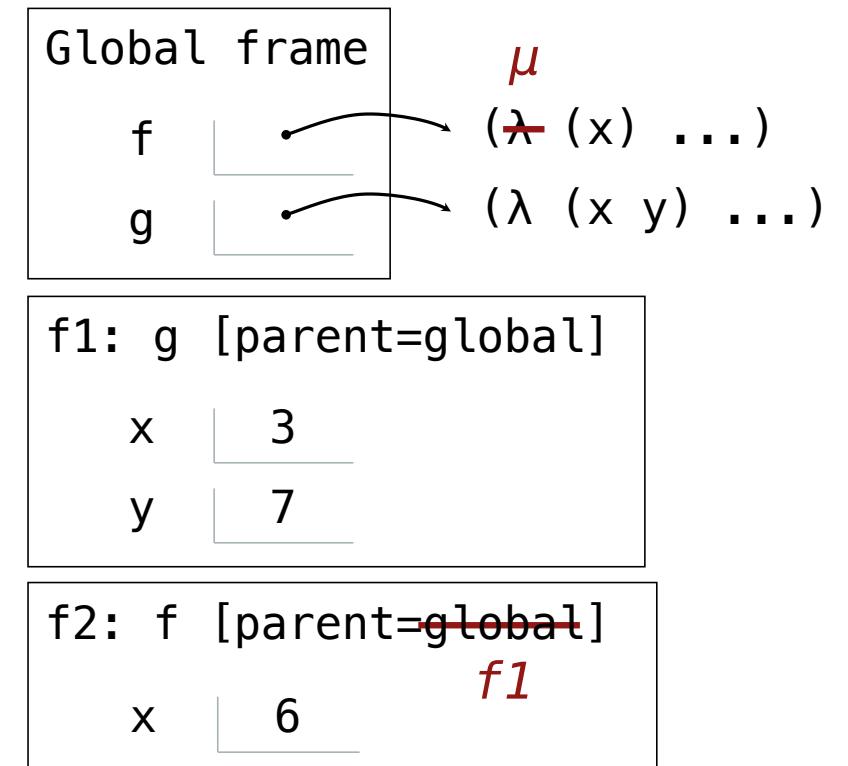
Dynamic scope: The parent of a frame is the environment in which a procedure was *called*

Special form to create dynamically scoped procedures (*mu* special form only exists in Project 4 Scheme)

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame
Error: unknown identifier: y

Dynamic scope: The parent for f's frame is g's frame



Tail Recursion

Functional Programming

All functions are pure functions

No re-assignment and no mutable data types

Name-value bindings are permanent

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated
- Sub-expressions can safely be evaluated in parallel or only on demand (lazily)
- **Referential transparency:** The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression

But... no `for/while` statements! Can we make basic iteration efficient? Yes!

Recursion and Iteration in Python

In Python, recursive calls always create new active frames

`factorial(n, k)` computes: $n! * k$

Time	Space
$\Theta(n)$	$\Theta(n)$
$\Theta(n)$	$\Theta(1)$

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```



```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

Tail Recursion

From the Revised⁷ Report on the Algorithmic Language Scheme:

"Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```
(define (factorial n k)
  (if (zero? n) k
      (factorial (- n 1)
                 (* k n))))
```

Should use resources like

How? Eliminate the middleman!

```
def factorial(n, k):  
    while n > 0:  
        n, k = n-1, k*n  
    return k
```

(Demo)

Time

Space

$\Theta(n)$

$\Theta(1)$

Tail Calls

A procedure call that has not yet returned is **active**. Some procedure calls are **tail calls**. A Scheme interpreter should support an **unbounded number** of active tail calls using only a **constant** amount of space.

A tail call is a call expression in a tail context:

- The last body sub-expression in a **lambda** expression (or procedure definition)
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
                 (* k n)))))
```

Example: Length of a List

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

The code is shown with a dashed blue box around the entire body of the `length` procedure. Inside this box, another dashed blue box surrounds the `(length (cdr s))` part of the `(+ 1 ...)` expression. A red rounded rectangle highlights this inner expression. A grey callout bubble points to it with the text "Not a tail context".

A call expression is not a tail call if more computation is still required in the calling procedure. This is called **embedded recursion**.

Linear recursive procedures can often be re-written to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n)))))
```

The code is shown with a dashed blue box around the entire body of the `length-iter` procedure. Inside this box, another dashed blue box surrounds the `(length-iter (cdr s) (+ 1 n))` part of the `(length-iter ...)` expression. A grey callout bubble points to it with the text "Recursive call is a tail call".

Eval with Tail Call Optimization (TCO)

The return value of the tail call is the return value of the current procedure call

Therefore, tail calls shouldn't increase the environment size

(Demo)

Examples: Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.

```
(define (length s)
  (+ 1 (if (null? s)
            -1
            (length (cdr s))))) )
```

;; Return the nth Fibonacci number.

```
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                      (fib (- k 1)))
                  (+ k 1))
                  ) )
    (if (= 1 n) 0 (fib-iter 1 2))))
```

;; Return whether s contains v.

```
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v)))))
```

;; Return whether s has any repeated elements.

```
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s))))) )
```

Map and Reduce

Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
              (cdr s)
              (procedure start (car s)))))
```

Recursive call is a tail call

Space depends on what `procedure` requires

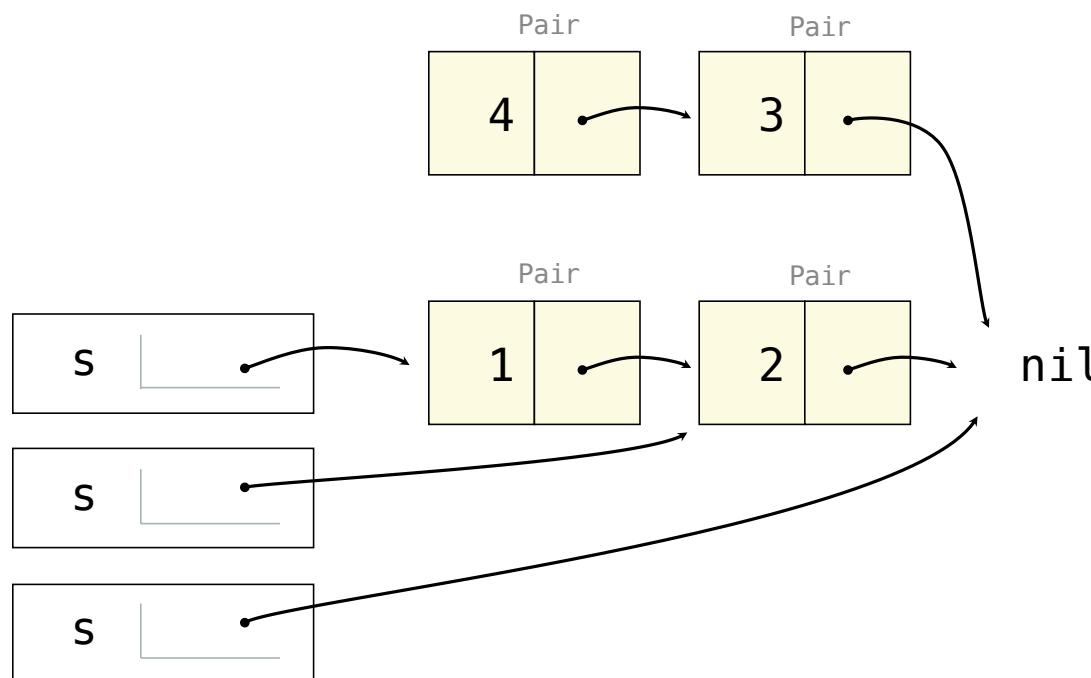
(`reduce *` '(3 4 5) 2) 120

(`reduce (lambda (x y) (cons y x))` '(3 4 5) '(2)) (5 4 3 2)

Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))) )
```

```
(map (lambda (x) (- 5 x)) (list 1 2))
```



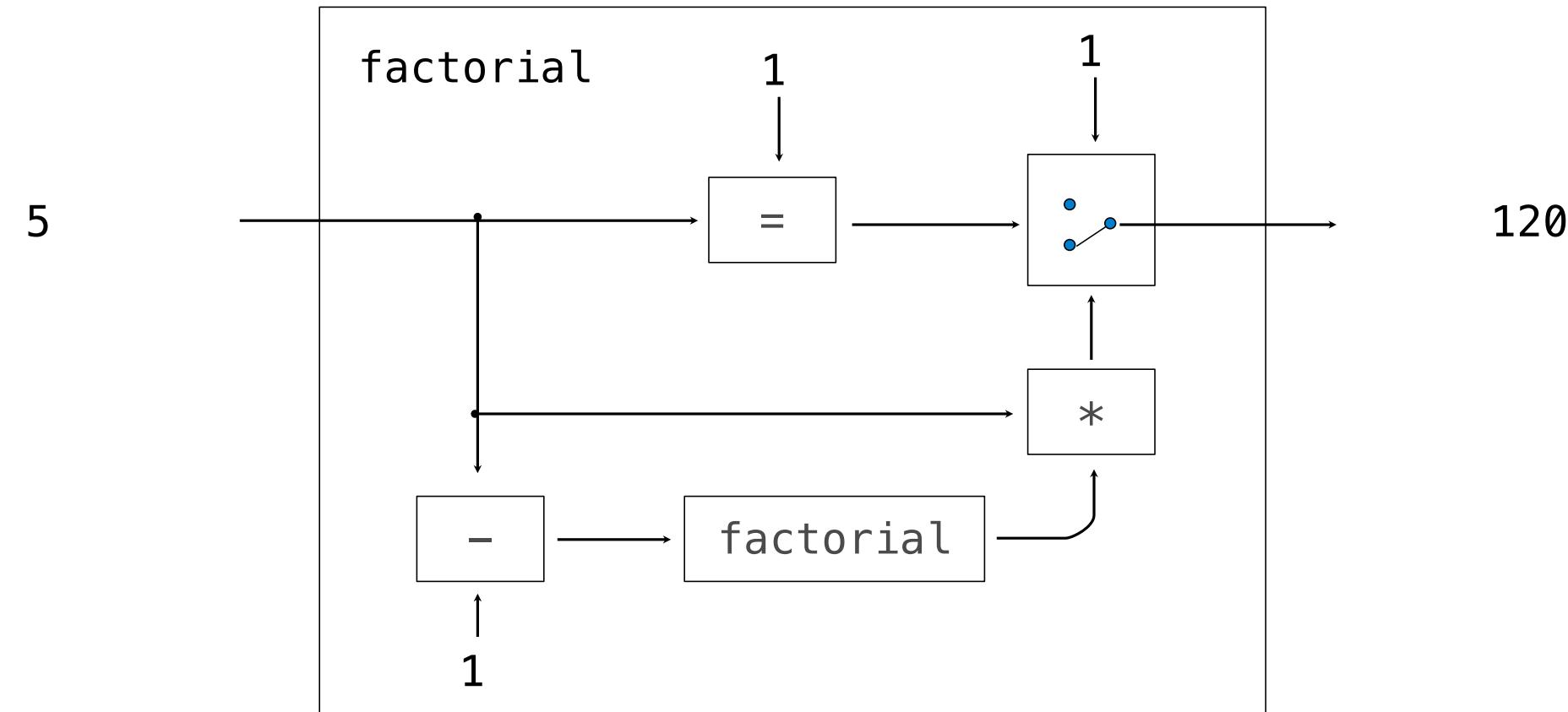
```
(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s)
        m
        (map-reverse (cdr s)
                     (cons (procedure (car s))
                           m)))) )
  (reverse (map-reverse s nil)))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                     (cons (car s) r)) ) )
  (reverse-iter s nil))
```

General Computing Machines

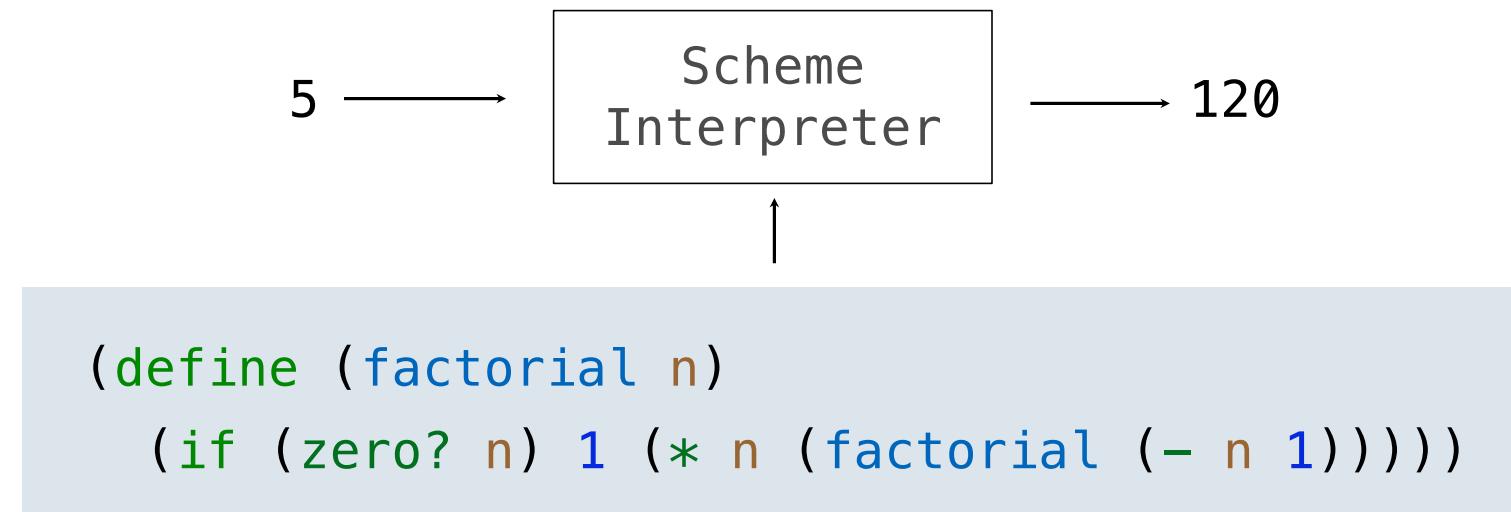
An Analogy: Programs Define Machines

Programs specify the logic of a computational device



Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine



Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of evaluation rules