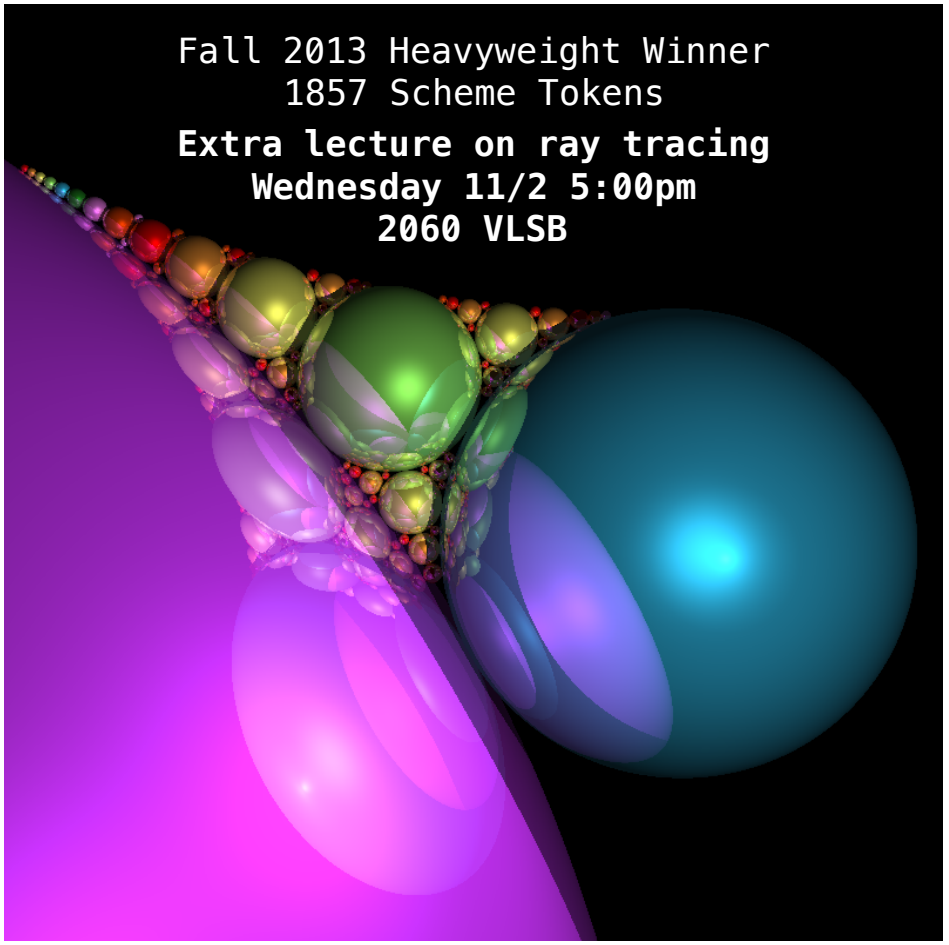
61A Lecture 28

Announcements

Scheme Recursive Art Contest: Start Early!

Fall 2012 Featherweight Winner
176 Scheme Tokens



Fall 2013 Heavyweight Winner
1857 Scheme Tokens

Extra lecture on ray tracing
Wednesday 11/2 5:00pm
2060 VLSB

Dynamic Scope

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)

mu

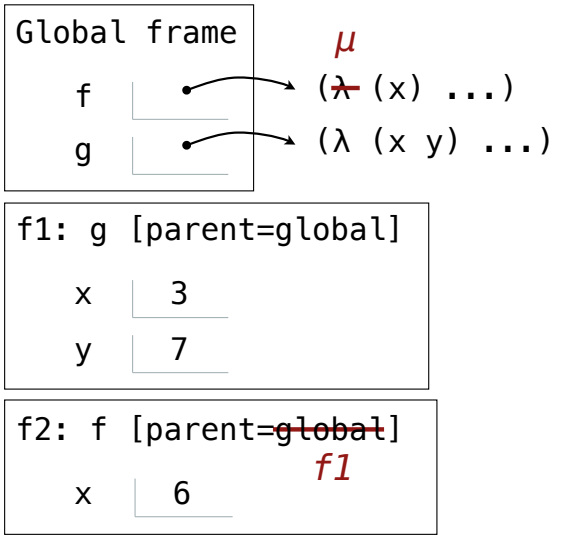
```
(define f (lambda (x) (+ x y)))

(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

13



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: $n! * k$

```
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
```

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

Tail Recursion

From the Revised[7] 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

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

How? Eliminate the middleman!

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

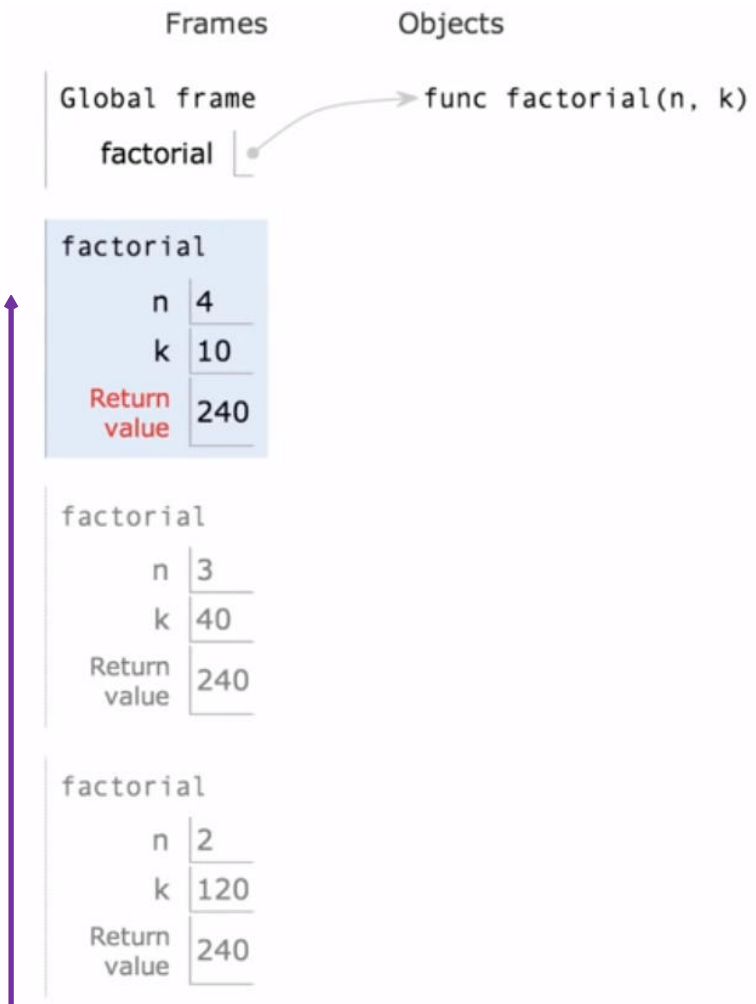
(Demo)

See next page.

Recursive frames are no longer needed after recursive calls are made.

simply passing the result

Tail Calls



and frames of
n=1
n=0, where we get the result

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
- 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))) ))
```

Not a tail context

A call expression is not a tail call if more computation is still required in the calling procedure

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))) )
  (length-iter s 0) )
```

Recursive call is a tail call

Eval with Tail Call Optimization

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

Unopitimized, call factorial 1000

```
  327   (factorial 673 65696325288404503575796625978978448712093
05286295455125585627590587109437571514935092066085412408456466 70
62644732678335137777116996174135041803126916882583093414876369 01
34072335800483410438362879044931997238461919779992353154366615 40
08350802387087285647792632804418417287598784791800766887276184 66
23413733512018842264707960761303823850445225810967246489524212 81
42950143075084568482609668437086931845081219573758732459995224 01
95876881510861709010772871656041765889356926330358792204394922 11
01870674245630601681096537743822810216254843547670945076429513 63
17278563654506335643671178593376044028801818532551845745469666 49
86875165612702560723373711057032382901505513672908167929970008 39
26944384515943640793137914535019286923066083847503993386673160 29
62087200011724323939821811776935257728545296471830615782661833 99
59868512221934741969570614203006068336032165495025217630843699 20
00000000000000000000000000000000000000000000000000000000000000 00
00000000000000000)
  328   (- 673 1)
Error: maximum recursion depth exceeded while calling a Python o
bject
```

error occurred when calling factorial 327,
using up all space for recursion

(Demo)

Optimized, call factorial 10000

```
scm> (factorial 10000 1)
28462596809170545189064132121198688901480514017027992307941799 94
27441134000376444377299078675778477581588406214231752883004233 99
40153518739052421161382716174819824199827592418289259787898124 25
31205946599625986706560161572036032397926328736717055741975962 09
94797203461536981198970926112775004841988454104755446424421365 73
30307670362882580354896746111709736957860367019107151273058728 10
```

...

Tail Recursion Examples

all recursive calls are tail calls

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)))))
```

already tail recursive, no worry

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

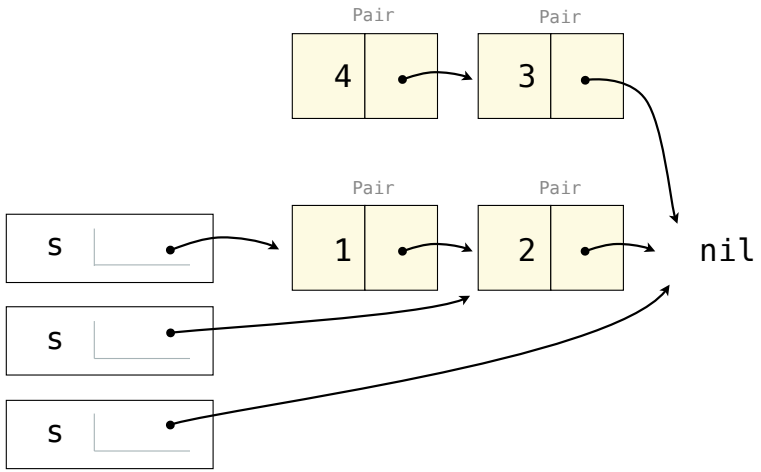
Space depends on what procedure requires

```
(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))
```



not tail recursive

```
(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)))
```

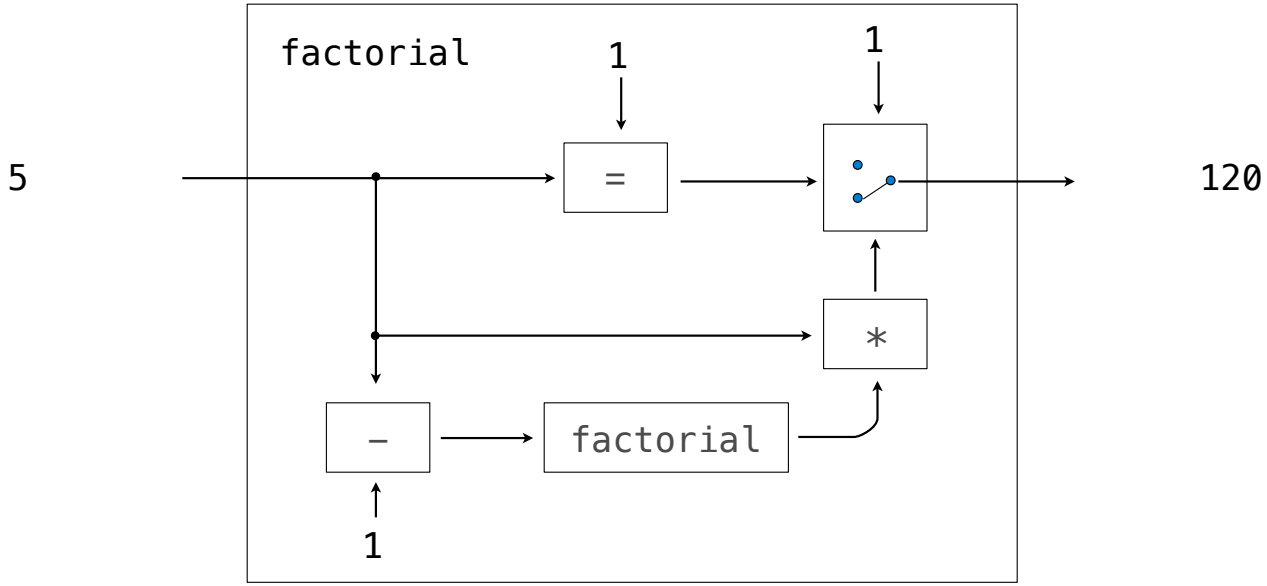
```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                      (cons (car s) r))))
  (reverse-iter s nil))
```

tail recursive

General Computing Machines

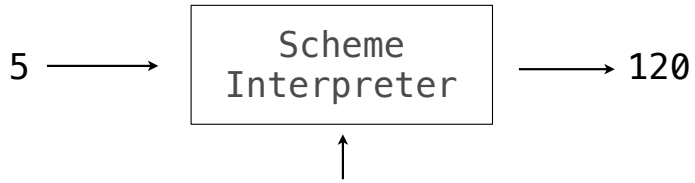
An Analogy: Programs Define Machines

Programs specify the logic of a computational device



Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine




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

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

An interpreter is nothing more than a program.