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

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

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

Lexical scope: The parent for f’s frame is the global frame

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

Error: unknown identifier: y

mu

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

Tail Recursion

In Python, recursive calls always create new active frames

Recursion and iteration in Python

factorial(n, k) computes: n! = k

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
</tbody>
</table>

// Time // Space

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

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

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!
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)))))

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.

- 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

Example: Length of a List

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

Not a tail context

(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

Tail Recursion Examples

Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? Θ(1)

1. Compute the length of s.
   Define (length s)
   \(|s| + (\text{if} \ (\text{null?} \ s) \ 0)\)
   (length (cdr s))

2. Return the nth Fibonacci number.
   Define (fib n)
   \(\text{if} \ (n = 0) \ k \ \text{fib-iter} \ current \ k\)
   \(\text{if} \ (n = 1) \ 0 \ \text{fib-iter} \ 1 \ 2\)

3. Return whether s contains v.
   Define (contains s v)
   \(\text{if} \ (\text{null?} \ s) \ false\)
   \(\text{if} \ (+ \ v \ (\text{car} \ s)) \ true\)
   \(\text{contains?} \ (\text{cdr} \ s) \ (v \ s)\)

4. Return whether s has any repeated elements.
   Define (has-repeat s)
   \(\text{if} \ (\text{null?} \ s) \ false\)
   \(\text{if} \ (\text{contains?} \ (\text{cdr} \ s) \ (\text{car} \ s)) \ true\)
   \(\text{has-repeat?} \ (\text{cdr} \ s) \ 1\)

Map and Reduce
Example: Reduce

```scheme
(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))`: 17

Example: Map with Only a Constant Number of Frames

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

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

(define 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))

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

Pair

- `(reduce (lambda (x y) (cons y x)) '(3 4 5) '2)`: `(5 4 3 2)`
- `(map lambda (x) (- x)) (list 1 2)`: `1`

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
  (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