Scheme II

Dynamic scoping, tail calls, and scheme practice
Announcements
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*

```
(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

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

```
Error: unknown identifier: y
```

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

```
mu
```

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

\[ \text{factorial}(n, k) \text{ computes: } n! \times k \]

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Linear</td>
</tr>
<tr>
<td>Linear</td>
<td>Constant</td>
</tr>
</tbody>
</table>
Tail Recursion

From the Revised 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."

\[
\text{(define (factorial n k)}
\text{(if (zero? n) k)}
\text{(factorial (- n 1)}
\text{(* k n)))}
\]

Should use resources like

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Constant</td>
</tr>
</tbody>
</table>

How? Eliminate the middleman!
Tail Recursion and Functional Programming

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

(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720

Example from: https://sarabander.github.io/sicp/html/1_002e2.xhtml#g_t1_002e2_002e1

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

(factorial 6 1)
(factorial 5 6)
(factorial 4 30)
(factorial 3 120)
(factorial 2 360)
(factorial 1 720)
720
Tail Calls
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)) ))
```

```
(define factorial (lambda (n k)
  (if (= n 0) k
      (factorial (- n 1) (* k n)) )))
```
Example: Length of a List

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

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))
      (length-iter s 0)))
```

Not a tail context

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

(Demo)
Tail Recursion Examples
Audience Participation
Is Length Tail Recursive?

Does this procedure run in constant space?

;;; Compute the length of s.
(define (length s)
  (+ 1
     (if (null? s)
         -1
         (length (cdr s))))
)

(length '(1 2 3))
Does this procedure run in constant space?

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v)))))

(contains '(1 2 3) 3)
Is Has-repeat Tail Recursive?

Does this procedure run in constant space?

;; 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)))))
Is fib Tail Recursive?

Which of the following procedures run in constant space?

;; 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))
    )
  )

 Đình tour: (Demo)
Tail recursive fib
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space?

;; 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))
              (+ 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))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space?

;; Compute the length of s.
(define (length s)
  (+ 1
    (if (null? s)
        -1
        (length (cdr s)))))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; 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 has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (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))))

;; Compute the length of s.
(define (length s)
  (+ 1
    (if (null? s)
        -1
        (length (cdr s)))))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; 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 has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s))))

;; Compute the length of s.
(define (length s)
  (+ 1
    (if (null? s)
        -1
        (length (cdr s)))))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; 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 has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s))))
Break

(Demo)
More turtle things
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)) (5 4 3 2)
```
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))))
)

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

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

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

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

(reverse (map (lambda (x) x) (list 1 2)))
```
Implementing Tail Call Optimization
**Who'da Thunk?**

**Thunk**: An expression wrapped in an argument-less function.

```python
>>> thunk1 = lambda: 2 * (3 + 4)
>>> thunk2 = lambda: add(2, 4)
>>> thunk1()
14
>>> thunk2()
6
```

Known as **Unevaluated** objects in the Scheme project.
**Trampolining**

**Trampoline**: A loop that iteratively invokes thunk-returning functions.

```python
def trampoline(f, *args):
    v = f(*args)
    while callable(v):
        v = v()
    return v
```

The function needs to be **thunk-returning**:

```python
def fact_k_thunked(n, k):
    if n == 0:
        return k
    return lambda: fact_k_thunked(n - 1, n * k)
```

```
trampoline(fact_k_thunked, 3, 1)
```

This way of executing the factorial function uses a constant number of frames.

Trampolining can simulate tail call optimization in unoptimized languages (e.g. Python).

(Demo)
Scheme Practice
Even Subsets

**Definition:** a *non-empty subset* of a list $s$ is a list containing some of the elements of $s$.

(A *non-empty subset* could contain all the elements of $s$, but not none of them.)

```scheme
;;; Non-empty subsets of integer list $s$ that have an even sum
;;;
;;; scm> (even-subsets '(3 4 5 7))
;;; ((5 7) (4 5 7) (4) (3 7) (3 5) (3 4 7) (3 4 5))
```

A recursive approach: The even subsets of $s$ include...

- all the even subsets of the rest of $s$
- the first element of $s$ followed by an (even/odd) subset of the rest
- just the first element of $s$ if it is even

(Demo)
Discussion Question: Even Subsets Using Filter
Discussion Question: Complete this implementation of even-subsets

Definition: a non-empty subset of a list \( s \) is a list containing some of the elements of \( s \).

(A non-empty subset could contain all the elements of \( s \), but not none of them.)

;;; non-empty subsets of \( s 
(define (nonempty-subsets s)
  (if (null? s) nil

      (let((rest (nonempty-subsets (cdr s)))))
        (append rest
            (map lambda (cons (car s) t)) rest)
            (list (list (car s))))))))

;;; non-empty subsets of integer list \( s \) that have an even sum
(define (even-subsets s)
  (filter (lambda (s) (even? (apply + s))) (nonempty-subsets s)))
Extra Tail Recursion Examples
Is camel Tail Recursive?

Does this procedure run in constant space?

;; Return whether n is a camel sequence. Ex: 121, 4142, 6590

define (camel n)
  (define (camel-helper n incr)
    (cond
      ((< n 10) #t)
      ((and (not incr) (camel-helper (quotient n 10) #t))
       (< (modulo (quotient n 10) 10) (modulo n 10)))
      ((and incr (camel-helper (quotient n 10) #f))
       (> (modulo (quotient n 10) 10) (modulo n 10)))
      (or (camel-helper n #t) (camel-helper n #f))))
Is camel Tail Recursive Now?

Does this procedure run in constant space?

`; Return whether n is a camel sequence. Ex: 121, 4142, 6590
(define (camel n)
  (define (camel-helper n incr)
    (cond
     ((< n 10) #t)
     (incr
      (and
       (camel-helper (quotient n 10) (not incr))
       (< (modulo (quotient n 10) 10) (modulo n 10))))
     (else
      (and
       (camel-helper (quotient n 10) (not incr))
       (> (modulo (quotient n 10) 10) (modulo n 10))))
     (or (camel-helper n #t) (camel-helper n #f))))

Is camel Tail Recursive Now?
Is camel Tail Recursive Now??

Does this procedure run in constant space?

;; Return whether n is a camel sequence. Ex: 121, 4142, 6590
(define (camel n)
  (define (camel-helper n incr)
    (cond
      ((< n 10) #t)
      (incr
       (and
        (< (modulo (quotient n 10) 10) (modulo n 10))
        (camel-helper (quotient n 10) (not incr))))
      (else
       (and
        (> (modulo (quotient n 10) 10) (modulo n 10))
        (camel-helper (quotient n 10) (not incr))))))
  (or (camel-helper n #t) (camel-helper n #f)))