Tail Calls
Class outline:

- Lexical vs. dynamic scopes
- Recursion efficiency
- Tail recursive functions
- Tail call optimization
Scopes
Lexical scope

The standard way in which names are looked up in Scheme and Python.

Lexical (static) scope: The parent of a frame is the frame in which a procedure was defined

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

Global frame

<table>
<thead>
<tr>
<th></th>
<th>f → λ (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>g → λ (x, y)</td>
</tr>
</tbody>
</table>

What happens when we run this code?

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td>7</td>
</tr>
</tbody>
</table>

|       | x | 6 |

f1: g [parent=Global]

f2: f [parent=Global]
Lexical scope

The standard way in which names are looked up in Scheme and Python.

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

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

---

Global frame

```
<table>
<thead>
<tr>
<th>f</th>
<th>x 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>y 7</td>
</tr>
</tbody>
</table>
```

f1: g [parent=Global]

```
| f | x 6 |
```

What happens when we run this code?
Error: unknown identifier: y
Dynamic scope

An alternate approach to scoping supported by some languages.

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

Scheme includes the `mu` special form for dynamic scoping.

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

Global frame

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>μ (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td></td>
<td>λ (x, y)</td>
</tr>
</tbody>
</table>

What happens when we run this code?

f1: g [parent=Global]

<table>
<thead>
<tr>
<th></th>
<th>x 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y 7</td>
<td></td>
</tr>
</tbody>
</table>

f2: f [parent=f1]

<table>
<thead>
<tr>
<th></th>
<th>x 6</th>
</tr>
</thead>
</table>
Dynamic scope

An alternate approach to scoping supported by some languages.

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

Scheme includes the `mu` special form for dynamic scoping.

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

What happens when we run this code?
13
Recursion efficiency
## Recursion and iteration in Python

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| ```python
def factorial(n, k):
    while n > 0:
        n = n - 1
        k = k * n
    return k
``` | | |
| ```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
``` | | |
Recursion and iteration in Python

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| ```python
def factorial(n, k):
    while n > 0:
        n = n - 1
        k = k * n
    return k
``` | Linear |       |
| ```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
``` | Linear |       |
# Recursion and iteration in Python

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| def factorial(n, k):
  while n > 0:
    n = n - 1
    k = k * n
  return k | Linear | Constant |

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| def factorial(n, k):
  if n == 0:
    return k
  else:
    return factorial(n-1, k*n) | Linear | Constant |
Recursion and iteration in Python

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| ```python
def factorial(n, k):
    while n > 0:
        n = n - 1
        k = k * n
    return k
``` | Linear    | Constant  |
|------|----------|-----------|
| ```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
``` | Linear    |           |
# Recursion and iteration in Python

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
| def `factorial`(n, k):
  while n > 0:
    n = n - 1
    k = k * n
  return k | Linear | Constant |
| def `factorial`(n, k):
  if n == 0:
    return k
  else:
    return `factorial`(n-1, k*n) | Linear | Linear |
Recursion frames in Python

In Python, recursive calls always create new frames.

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

Active frames over time:

![Diagram showing recursive frames and their values over time.]

View in PythonTutor
Recursion in Scheme

In Scheme interpreters, a tail-recursive function should only require a constant number of active frames.

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

Active frames over time:
Tail recursive functions
Tail recursive functions

In a **tail recursive function**, every recursive call must be a tail call.

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

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**
Example: Length of list

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

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

But linear recursive procedures can often be re-written to use tail calls...
Example: Length of list

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

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

But 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) )
Is it tail recursive?

;;; 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))))
Is it tail recursive?

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

❌ No, because **if** is not in a tail context.

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

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

✗ No, because if is not in a tail context.

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

✓ Yes, because contains is in a tail context if.
;;; 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)))
Is it tail recursive? 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)))))

✓ Yes, because has-repeat is in a tail context.

;; 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)))
Is it tail recursive? 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)))))

✔ Yes, because has-repeat is in a tail context.

;;; 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)))

❌ No, because fib is not in a tail context.
Example: Reduce

\[
\begin{align*}
\text{reduce} & \quad * \quad '(3 \ 4 \ 5) \ 2 \ 120 \\
\text{reduce} & \quad (\text{lambda} \ (x \ y) \ (\text{cons} \ y \ x)) \quad '(3 \ 4 \ 5) \quad '(2) \quad (5 \ 4 \ 3 \ 2)
\end{align*}
\]
Example: Reduce

(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)

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

Is it tail recursive?
Example: Reduce

(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)

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

Is it tail recursive?

✔ Yes, because reduce is in a tail context.
Example: Reduce

\[(\text{reduce } \ast \ '(3 \ 4 \ 5) \ 2) \ 120\]
\[(\text{reduce } (\lambda(x \ y) (\text{cons } y \ x)) \ '(3 \ 4 \ 5) \ '(2)) \ (5 \ 4 \ 3 \ 2)\]

\[(\text{define } (\text{reduce } \text{procedure } s \ \text{start})\]
\[\quad (\text{if } (\text{null? } s) \ \text{start}\]
\[\quad (\text{reduce } \text{procedure}\]
\[\quad \quad (\text{cdr } s)\]
\[\quad \quad (\text{procedure } \text{start} \ (\text{car } s)) \)) \))\]

Is it tail recursive?

✓ Yes, because \text{reduce} is in a tail context.

However, if \text{procedure} is not tail recursive, then this may still require more than constant space for execution.
Example: Map

(map (lambda (x) (- 5 x)) (list 1 2))
Example: Map

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

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

Is it tail recursive?
Example: Map

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

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

Is it tail recursive?
❌ No, because `map` is not in a tail context.
Example: Map (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))

(map (lambda (x) (- 5 x)) (list 1 2))
Tail call optimization with trampolining
What the thunk?

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

Making thunks in Python:

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

Calling a thunk later:

```python
thunk1()
thunk2()
```
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! One possibility:

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

```python
trampoline(factorial_thunked, 3, 1)
```

[View in PythonTutor]
Demo: Trampolined interpreter

The Scheme project EC is to implement trampolining. Let's see how it improves the ability to call tail recursive functions...