CS 152
Programming Paradigms

More Scheme

Today

- Higher Order Functions
- Recursion and Efficiency
- Tail Recursion

Higher Order Functions

- Functions that take other functions as parameters
- Functions that return functions as values

Example: filter

Goal: write a function *filter,* which takes a predicate (function) p and a list as parameters, and returns a new list containing only elements of the original list for which p evaluates to #t.

```
(filter integer? '(6 3.4 "hello" 4 2.3 #t))
(6 4)
(filter string? '(6 3.4 "hello" 4 2.3 #t))
("hello")
(filter string? '())
()
(filter (lambda(x) (>= x 5)) '(6 3 8 2))
(6 8)
```

Example: filter

```
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to  #t.
(define
  (filter p xs)
```

base case?
(null? xs)   -> '()

Example: filter

```
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to #t.
(define
  (filter p xs)
```

Recursive rule?
How do we compute (filter p xs) assuming that we know how to compute (filter p (cdr xs))

Example: filter

```
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to #t.
(define
  (filter p xs)
```

Recursive rule?
We need to evaluate (p (car xs)).
If true: include (car xs) in new result:
(cons (car xs) (filter p (cdr xs)))
If false ignore (car xs):
(filter f (cdr xs)))

Example: filter

```
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to #t.
(define (filter p xs)
  (cond ((null? xs) '()) ; base case
        ((p (car xs)) (cons (car xs) (filter p (cdr xs)))) ; include car
        (else (filter p (cdr xs))))) ; ignore car
```

map

The map function is a built-in function that takes a function and one or more lists as parameters. It applies the function to every element of the list(s), and returns the list of the results.

```
>(map abs '(-2 3 5 -1.4))
(2 3 5 1.4)
>(map square '(0 2 3))
(0 4 9)
> (map + '(1 2 3) '(4 5 6))
'(5 7 9)
```

The lists must have the same size.

Recursion and Efficiency

Recursion and Efficiency in Python

Canvas -> Resources -> Recursive Functions in Python

```
def factorial(n):
   if n <= 1:
         return 1
    else:
        return n * factorial(n-1)
print(factorial(5))
120
```

Recursion and Efficiency in Python

Canvas -> Resources -> Recursive Functions in Python

```
def factorial(n):
   if n <= 1:
         return 1
    else:
        return n * factorial(n-1)
print(factorial(500))
```

1220136825991110068701238785423046926253574342803192842192413588385845
37315.....8283557801587354327688886801203998823847021514676054454076635 3
5984174430480128938313896881639487469658817504506926365338175055478128
6400000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000

Recursion and Efficiency in Python

```
def factorial(n):
  if n <= 1:
      return 1
   else:
     return n * factorial(n-1)
print(factorial(1000))
```

RecursionError: maximum recursion depth exceeded ...

Iterative Implementation

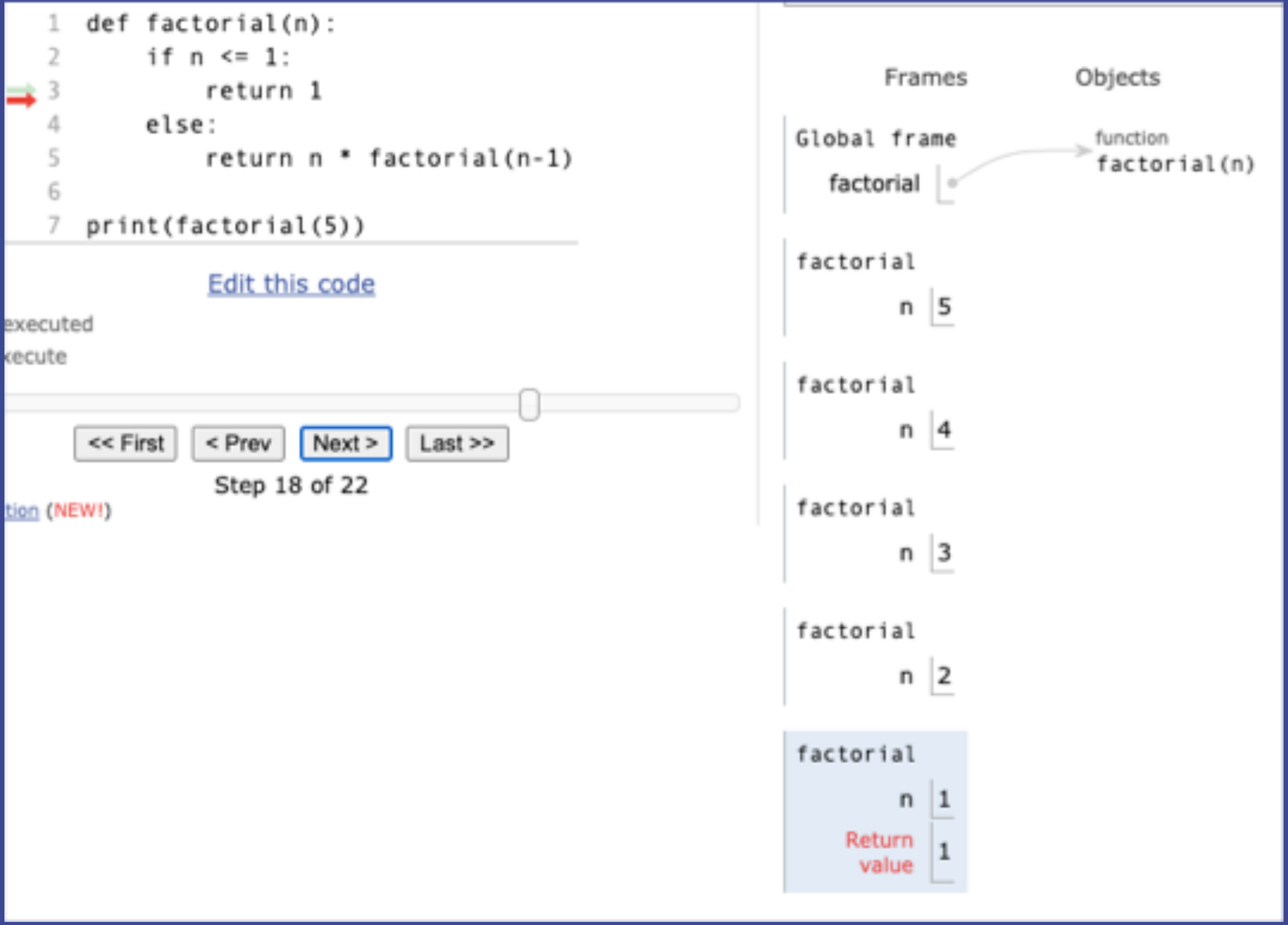
```
def loop_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
```

```
print(loop_factorial(1000))
print(loop_factorial(50000))
402387260077093773543702...
334732050959714483691547...
```

Visualizing Recursive Calls

- Canvas -> Resources -> Visualizing Recursive Implementation
- Python Visualizer: http://pythontutor.com/

Visualizing Recursive Calls



Recursion and Efficiency

In Python, recursive calls always create new active frames

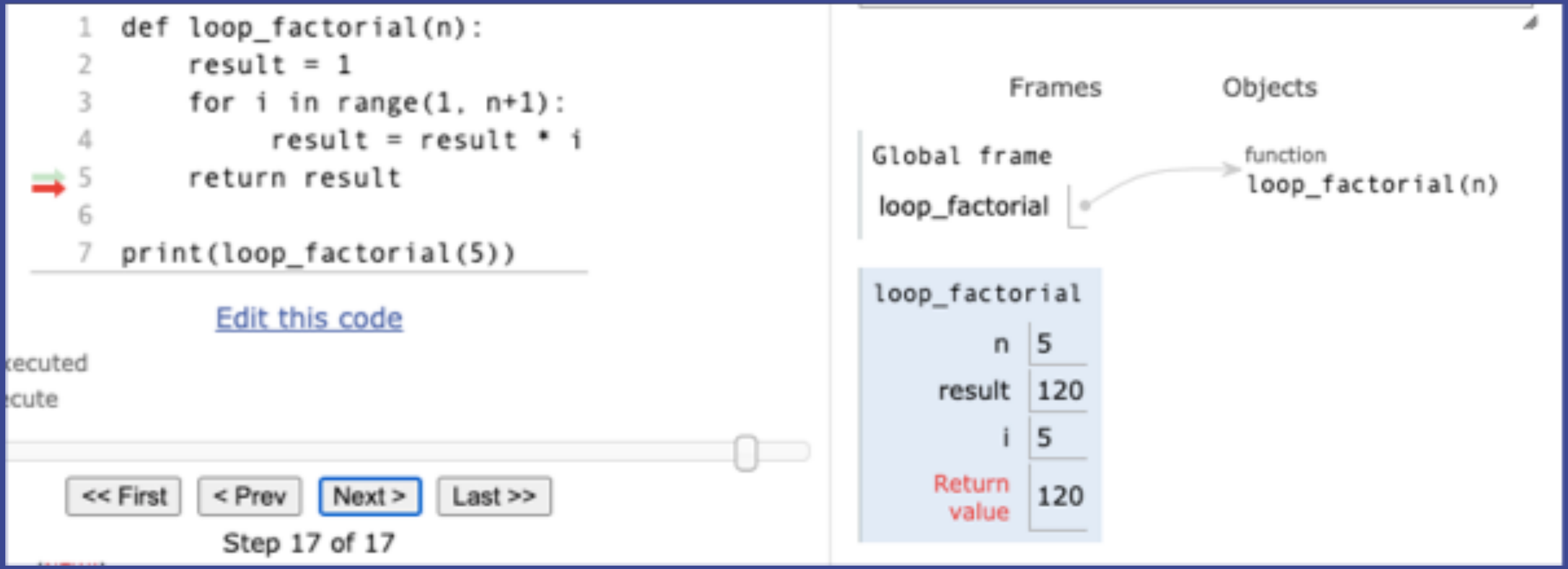
```
def factorial(n):
   if n <= 1:
         return 1
    else:
        return n * factorial(n-1)
 print(factorial(5))
```

Complexity?

Time complexity: O(n)
Space complexity: O(n)

Visualizing Iterative Implementation

- Canvas -> Resources -> Visualizing Iterative Implementation



Iterative Implementation

```
def loop_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result

print(loop_factorial(5))
```

Time complexity: O(n)
Space complexity: O(1)
Because of that, a non-recursive implementation is usually preferable in Python.

Scheme and Recursion

"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."

Tail Calls

- Some function calls are tail calls. They represent the final action in the caller.
- A function call is not a tail call if more computation is required in the caller.
- A function call that has not yet returned is active.
- A Scheme interpreter supports an unbounded number of active tail calls using only a constant amount of space.

iClicker: Tail Call?

```
def factorial(n):
  if n <= 1:
      return 1
   else:
     return n * factorial(n-1)
```

A. Yes - final action in the caller
B. No - more computation is required in the caller.

iClicker: Tail Call?

```
def factorial(n, result=1):
    if n <= 1:
        return result
    else:
        return factorial(n-1, result * n)
```

A. Yes - final action in the caller
B. No - more computation is required in the caller.

Tail Call Optimization

Python does not optimize tail calls.

```
def factorial(n, result=1):
    if n <= 1:
        return result
    else:
        return factorial(n-1, result * n) # factorial is tail recursive
> print(factorial(1000, 1))
RecursionError: maximum recursion depth exceeded
```

Making an effort to write tail recursive functions in Python does not make any difference.

Scheme optimizes tail calls. It supports an unbounded number of active tail calls using only a constant amount of space.

Constant Space – O(1)?

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

A. Yes
B. No

Tail Call?

```
;;; Function count: element list -> number
;;; Returns the count of the given element in the list
(define (count x xs)
        (cond ((null? xs) 0)
              ((equal? x (car xs)) (+ 1 (count x (cdr xs))))
              (else (count x (cdr xs)))))
```

A. Yes

B. No

Constant Space – O(1)?

```
;;; Function count: element list -> number
;;; Returns the count of the given element in the list
(define (count x xs)
        (cond ((null? xs) 0)
              ((equal? x (car xs)) (+ 1 (count x (cdr xs))))
              (else (count x (cdr xs)))))
```

A. Yes

B. No

Tail Call?

```
;;; Predicate all-positive?: list of numbers -> boolean
;;; Returns #t if no element is less than 0 and #f otherwise
(define (all-positive? xs)
  (if (null? xs)
      #t
      (and (>= (car xs) 0 ) (all-positive? (cdr xs)))))
```

But before we answer this question…

Short Circuit Evaluation

Scheme function parameters are evaluated at the time the function is called (applicative order evaluation, pass by value).

(f (+ 1 3) (* 4 5)): (+ 1 3) and (* 4 5) are first evaluated then f is called with the results: (f 4 20)

However 'and' and 'or' are not functions.

They are special forms.

They implement a short circuit evaluation.

Short Circuit Evaluation

(and A B):

- B is only evaluated if A is true
- If A is true, B is returned

(or C D):

- D is only evaluated if C is false
- If C is false, D is returned

Tail Call?

```
;;; Predicate all-positive?: list of numbers -> boolean
;;; Returns #t if no element is less than 0 and #f otherwise
(define (all-positive? xs)
  (if (null? xs)
      #t
      (and (>= (car xs) 0 ) (all-positive? (cdr xs)))))
```

A. Yes
B. No

Tail Recursion

Linear recursive functions can often be rewritten to use tail calls.

- Turn the original function into a helper function.
- Add an accumulator argument to the helper function.
- Update the base case.
- Change the helper function's recursive call into a tail-recursive call. The accumulator must be updated.
- Make the body of the main function just a call to the helper, with appropriate initial values of the accumulator.

How do we turn it into tail recursion?

```
;;; Function count: element list -> number
;;; Returns the count of the given element in the list
(define (count x xs)
        (cond ((null? xs) 0)
              ((equal? x (car xs)) (+ 1 (count x (cdr xs))))
              (else (count x (cdr xs)))))
```

Turn the original function into a helper function.

How do we turn it into tail recursion?

```
(define (tcount x xs )
        (cond ((null? xs) 0)
              ((equal? x (car xs)) (+ 1 (tcount x (cdr xs))))
              (else (tcount x (cdr xs)))))
```

Add an accumulator argument to the helper function.

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
        (cond ((null? xs) 0)
              ((equal? x (car xs)) (+ 1 (tcount x (cdr xs))))
              (else (tcount x (cdr xs)))))
```

What is the type of *count-so-far*?

A. number
B. string
C. list

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
    (cond ((null? xs) 0)
          ((equal? x (car xs)) (+ 1 (tcount x (cdr xs))))
          (else (tcount x (cdr xs)))))
```

Update the base case?

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
   (cond ((null? xs)  count-so-far)
         ((equal? x (car xs)) (+ 1 (tcount x (cdr xs))))
         (else (tcount x (cdr xs)))))
```

Update the base case?

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
        (cond ((null? xs) count-so-far)
              ((equal? x (car xs)) (+ 1 (tcount x (cdr xs))))
              (else (tcount x (cdr xs)))))
```

Change the helper function's recursive call into a tail-recursive call. The accumulator must be updated.

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
    (cond ((null? xs) count-so-far)
            ((equal? x (car xs)) (tcount x (cdr xs) (+ 1 count-so-far)))
            (else (tcount x (cdr xs) count-so-far))))
```

How do we turn it into tail recursion?

```
(define (tcount x xs count-so-far)
   (cond ((null? xs) count-so-far)
         ((equal? x (car xs)) (tcount x (cdr xs) (+ 1 count-so-far)))
         (else (tcount x (cdr xs) count-so-far))))
(define (count x xs) (tcount x xs 0))
```

Make the body of the main function just a call to the helper, with appropriate initial values of the accumulator.

Tail Recursive *count*

```
;;; Function tcount: element list number -> number
;;; Tail recursive helper function
(define (tcount x xs count-so-far)
        (cond ((null? xs) count-so-far)
              ((equal? x (car xs)) (tcount x (cdr xs) (+ 1 count-so-far)))
              (else (tcount x (cdr xs) count-so-far))))
;;; Function count: element list -> number
;;; Returns the count of the given element in the list
(define (count x xs) (tcount x xs 0))
```

Tail Recursive *filter*

```
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to #t.
(define (filter p xs)
  (cond ((null? xs) '()) ; base case
        ((p (car xs)) (cons (car xs) (filter p (cdr xs)))) ; include car
        (else (filter p (cdr xs))))) ; ignore car
```

Turn the original function into a helper function.

Tail Recursive *filter*

```
(define
  (tfilter p xs)
  (cond
    ((null? xs) '()) ; base case
    ((p (car xs)) (cons (car xs) (tfilter p (cdr xs)))) ; include car
    (else (tfilter p (cdr xs))))) ; ignore car
```

Add an accumulator argument to the helper function.

Tail Recursive *filter*

```
(define
  (tfilter p xs sofar)
  (cond
    ((null? xs) '()) ; base case
    ((p (car xs)) (cons (car xs) (tfilter p (c
    (else (tfilter p (cdr xs))))) ; ignore car
```

What is the type of the accumulator *sofar*?

A. number
B. string
C. list

Tail Recursive *filter*

Update the base case?

```
(define
 (tfilter p xs sofar)
 (cond
  ((null? xs) '()) ; base case
  ((p (car xs)) (cons (car xs) (tfilter p (cdr xs)))) ; include car
  (else (tfilter p (cdr xs))))) ; ignore car
```

Tail Recursive *filter*

```
(define
 (tfilter p xs sofar)
 (cond
  ((null? xs) sofar) ; base case
  ((p (car xs)) (cons (car xs) (tfilter p (cdr xs)))) ; include car
  (else (tfilter p (cdr xs))))) ; ignore car
```

Update the base case?

Tail Recursive *filter*

```
(define
  (tfilter p xs sofar)
  (cond
    ((null? xs) sofar) ; base case
    ((p (car xs)) (cons (car xs) (tfilter p (cdr xs)))) ; include car
    (else (tfilter p (cdr xs))))) ; ignore car
```

Change the helper function's recursive call into a tail-recursive call. The accumulator must be updated.

Tail Recursive *filter*

```
(define
 (tfilter p xs sofar)
 (cond
  ((null? xs) sofar) ; base cas
  ((p (car xs)) (cons (car xs) (
  (else (tfilter p (cdr xs))))) ; ignore car
```

Can we use cons to update the accumulator? (cons (car xs) sofar)

A. No
B. Yes

Tail Recursive *filter*

```
(define
  (tfilter p xs sofar)
  (cond
    ((null? xs) sofar) ; base case
    ((p (car xs)) (tfilter p (cdr xs) (append sofar (list (car xs)))))
    (else (tfilter  p (cdr xs) sofar)))); ignore car
```

Make the body of the main function just a call to the helper, with appropriate initial values of the accumulator.

Tail Recursive *filter*

```
(define
  (tfilter p xs sofar)
  (cond
    ((null? xs) sofar) ; base case
    ((p (car xs)) (tfilter p (cdr xs) (append sofar (list (car xs)))))
    (else (tfilter  p (cdr xs) sofar)))); ignore car

(define (filter p xs) (tfilter p xs '()))
```

Tail Recursive *filter*

```
;;; Function tfilter: predicate list list -> list
;;; Tail recursive helper function
(define
  (tfilter p xs sofar)
  (cond
    ((null? xs) sofar) ; base case
    ((p (car xs)) (tfilter p (cdr xs) (append sofar (list (car xs)))))
    (else (tfilter  p (cdr xs) sofar)))); ignore car
;;; Function filter:  predicate list -> list
;;; Returns a new list containing only elements of the original list
;;; for which the predicate evaluates to #t.
(define (filter p xs) (tfilter p xs '()))
```

Tail Recursion

- Not every recursive function can be turned into a tail-recursive function.
- If a function makes a recursive call, then examines the result and does different things depending on its value, then it may not be possible to make the function tail-recursive.

Reminders

- Homework 3 due tomorrow by 5 PM.
- Exam 1: September 23
  - Take the practice quiz if you have not done so yet
- Next: Haskell