CS 61A SU26

Lecture 21: Tail Calls and Macros

July 28, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Lecture 20 Updates
- Announcements
- Computing in the News

Lecture 20 Updates (1 of 2)

- **Fixed last PollEv question** so it uses the correct screenshot with a `mu` procedure
- **Updated the GitHub repository** to have type hints, more comprehensive doctests, and update the docstring at the top of `calc.py` (diff)
- **I incorrectly stated** during lecture yesterday **that `calc_eval` only takes in `Link` objects**; it can take in any Scheme Calculator expression, which could be a Scheme list (`Link`) or other things like numbers or variables
- **I also incorrectly stated** that the **Scheme Calculator interpreter** we're implementing **is mutually recursive**. The **full Scheme interpreter** (final project) ***is*** mutually recursive, but because Scheme Calculator has no user-defined procedures, `calc_apply` never calls `calc_eval` (however, `calc_eval` ***does*** call `calc_apply`)

Lecture 20 Updates (2 of 2)

- Reminder: **The buggy behavior of 61A's `filter` function** (mentioned in Lec 19) **is unintentional**, **but you won't need to care** for the Scheme project and it is not currently a priority fix for the content/infra team.
  - **On an exam we will be clear that you should use the intended `filter` behavior** (exclude values for which calling the predicate function returns `#f`)
- Reminder: **Skipped slides** indicate we skipped during live lecture, but **they're still in scope** unless otherwise noted

Announcements

- Special topics will be: Computer Security, Web Applications, and AI Coding Tools (not necessarily in that order).
  - They will be "lightly in scope" for the final, which means we'll only have an extra credit question on the exam for them so please come if you can and are interested!

Computing in the News

- Some people's chats with Claude AI found publicly available online (BBC)
  - "Hundreds of user conversations with Anthropic's popular artificial intelligence (AI) chatbot Claude were found to have been available to essentially anyone online... **The searches showed Claude chats for which a user had decided to 'share' a link had been saved by search engines like Google, leaving them accessible to the broader public.** The search availability of the chat logs was removed over the weekend, but many were saved and shared widely online."
  - Discovered by Redditors 3 days ago; also happened in 2025
- Microsoft launches its first cybersecurity model, plus a new agentic cybersecurity system (TechCrunch)
- Apple plans to lease iPhones for $17.99 a month through partnership with Klarna (CNBC)

Review: Interpreters

10 min

- Eval-Apply
- Dynamic Scope

Eval-Apply PollEv Answer Options

(A)

```
1.  Eval((* 2 (+ 1 9) 4))
2.  Eval('*')
3.  Eval(2)
4.  Eval((+ 1 9))
5.  Apply('+', (1 9))
6.  Eval(4)
7.  Apply('*', (2 10 4))
```

(B)

```
1.  Eval((* 2 (+ 1 9) 4))
2.  Eval('*')
3.  Eval(2)
4.  Eval((+ 1 9))
5.  Eval('+')
6.  Eval(1)
7.  Eval(9)
8.  Apply('+', (1 9))
9.  Eval(4)
10. Apply('*', (2 10 4))
```

(C)

```
1.  Eval((* 2 (+ 1 9) 4))
2.  Eval(2)
3.  Eval((+ 1 9))
4.  Eval(1)
5.  Eval(9)
6.  Apply('+', (1 9))
7.  Eval(4)
8.  Apply('*', (2 10 4))
```

(D)

```
1.  Eval((* 2 (+ 1 9) 4))
2.  Eval('*')
3.  Eval(2)
4.  Eval((+ 1 9))
5.  Eval(4)
6.  Eval('+')
7.  Eval(1)
8.  Eval(9)
9.  Apply('+', (1 9))
10. Apply('*', (2 10 4))
```

(E)

```
1.  Apply('*', (2 10 4))
2.  Eval((* 2 (+ 1 9) 4))
3.  Eval('*')
4.  Eval(2)
5.  Apply('+', (1 9))
6.  Eval((+ 1 9))
7.  Eval('+')
8.  Eval(1)
9.  Eval(9)
10. Eval(4)
```

Scheme expression:

```
(* 2 (+ 1 9) 4)
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

**Select the correct chronological ordering of the Eval-Apply mutual recursion that happens in the full Scheme interpreter when evaluating the expression:
 (* 2 (+ 1 9) 4)
 For answer options, see slide 8.**

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Dynamic Scope: What would Scheme display?

Tail Calls

20 min

- Functional Programming
- Recursion and Iteration in Python
- Tail Recursion in Scheme
- Tail Recursion

Functional Programming

"Strictly" functional programming involves:

- Only pure functions (no side effects)
- No reassignment or mutation
- Permanent name-value bindings

Advantages:

- The value of an expression is independent of the order in which subexpressions are evaluated (in our Scheme interpreter implementation, we decided to evaluate from left to right, but we don't have to)
  - This means we can evaluate subexpressions **in parallel** or **on demand (lazily)**
- Calling a function with the same arguments produces the same results
  - Allows for **memoization**

Disadvantages:

- **No iteration** (for/while statements) could lead to **space complexity explosion** when we recurse (too many frames)

Recursion and Iteration in Python

In Python, recursive calls always create new active frames. (Python is not optimized for tail recursion).

Suppose we want to compute `factorial(n, k)` which is defined as `k * n!`

Recursive

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

Time Complexity: O(n)
Space Complexity: O(n)

See environment diagram

Iterative

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

Time Complexity: O(n)
Space Complexity: O(1)

Tail Recursion in Scheme

Unlike Python, Scheme ***is*** optimized for tail recursion, e.g. in the previous example, the equivalent Scheme `factorial` procedure would take O(1) space by eliminating unnecessary frames. **(Demo)**

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

Tail Recursion (1 of 3)

- Are all recursive functions in Scheme space-optimized? **No**, the function must be written in a **tail recursive** way to take advantage of this feature.

Let's formalize this:

- 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 (additional) space.
- A tail call is a call expression in a **tail context**:
  - The last body subexpression in a user-defined procedure (e.g. using `define` or `lambda`)
  - Subexpressions 2 and 3 (consequent and alternative) in a tail context `if`
  - All non-predicate subexpressions in a tail context `cond`
  - The last subexpression in a tail context `and` or `or`
  - The last subexpression in a tail context `begin`

Tail Recursion (2 of 3)

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

Last body subexpression of user-defined procedure

Consequent or alternative of `if` in a tail context

Tail Recursion (3 of 3)

- A call expression **is *not*** tail recursive if more computation is still required in the calling procedure after the recursive call has returned.
- Often, linear recursive procedures can be rewritten to be tail recursive.

```
(define (length lst)
    (if
        (null? lst)
        0
        (+ 1 (length (cdr lst)))
    )
)
```

After `length` returns, we still need to add 1 to the result (there's still work left to do)

```
(define (length lst)
    (define (helper lst n)
        (if
            (null? lst)
            n
            (helper (cdr lst) (+ 1 n))
        )
    )
    (helper lst 0)
)
```

We already performed the "add 1" operation when we evaluated the arguments of the `helper` recursive call

Tail Recursion PollEv Answer Options (1 of 2)

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

```
; (B) Return whether s contains v
(define (contains s v)
    (if (null? s)
        #f
        (if (= v (car s))
            #t
            (contains (cdr s) v)
        )
    )
)
```

Tail Recursion PollEv Answer Options (2 of 2)

```
; (C) Return the n-th Fibonacci number
(define (fib n)
    (define (helper current k)
        (if (= n k)
            current
            (helper
                (+ current (fib (- n 1)))
                (+ k 1)
            )
        )
    )
    (if (= n 1) 0 (helper 1 2))
)
```

```
; (D) Return whether s has any repeated elements
; (assume contains is defined as in (C))
(define (has-repeat s)
    (if (null? s)
        #f
        (if (contains (cdr s) (car s))
            #t
            (has-repeat (cdr s))
        )
    )
)
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Which of the following procedures are tail recursive? (Assume they're all correctly implemented.) See answer options on slides 18-19.

Tail Recursion PollEv Solution (1 of 2)

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

```
; (B) Return whether s contains v
(define (contains s v)
  (if (null? s)
      #f
      (if (= v (car s))
          #t
          (contains (cdr s) v)
      )
  )
)
```

Tail Recursion PollEv Solution (2 of 2)

```
; (C) Return the n-th Fibonacci number
(define (fib n)
    (define (helper current k)
        (if (= n k)
            current
            (helper
                (+ current (fib (- n 1)))
                (+ k 1)
            )
        )
    )
    (if (= n 1) 0 (helper 1 2))
)
```

```
; (D) Return whether s has any repeated elements
; (assume contains is defined as in (B))
(define (has-repeat s)
    (if (null? s)
        #f
        (if (contains (cdr s) (car s))
            #t
            (has-repeat (cdr s))
        )
    )
)
```

Macros

20 min

- Programs as Data
- Using Quotes to Write a Program Which Writes a Program
- Macros

Programs as Data

- Recall: In many languages, **you can treat programs as data** (not just steps to execute)
  - Higher-order functions
  - In Python, the `eval` function takes in an arbitrary string and executes it as Python code
  - In Scheme, the `eval` procedure takes in an arbitrary Scheme list and executes it as a Scheme expression
- In such languages, it is easy to **write a program that writes a program**

Using Quotes to Write a Program Which Writes a Program (1 of 3)

```
scm> (+ 1 2)
3
scm> (list + 1 2)
(#[+] 1 2)
scm> (list '+ 1 2)
(+ 1 2)
scm> (list '+ 1 (+ 2 3))
(+ 1 5)
```

Using Quotes to Write a Program Which Writes a Program (2 of 3)

```
; Returns n factorial
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))
    )
)
```

```
; Returns a Scheme expression
; that evaluates to n factorial
(define (fact-exp n)
    (if (= n 0)
        1
        (list '* n (fact-exp (- n 1))))
    )
)
```

```
scm> (fact 5)
120
scm> (fact-exp 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

Using Quotes to Write a Program Which Writes a Program (3 of 3)

```
; Returns the n-th Fibonacci number
(define (fib n)
    (if (<= n 1)
        n
        (+
            (fib (- n 2))
            (fib (- n 1))
        )
    )
)
```

```
; Returns a Scheme expression which
; evaluates to the n-th Fibonacci number
(define (fib-exp n)
    (if (<= n 1)
        n
        (list '+
            (fib-exp (- n 2))
            (fib-exp (- n 1))
        )
    )
)
```

```
scm> (fib 6)
8
scm> (fib-exp 6)
(+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1)) (+ (+ 0 1) (+ 1 (+ 0 1)))))
scm> (eval (fib-exp 6))
8
```

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Select all of the following where executing the Scheme code which would result in 2 and 2 being displayed, each on its own line.

Macros (1 of 9)

```
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (eval (twice '(print 2)))
2
2
scm> (eval (twice '(print "yeet")))
"yeet"
"yeet"
scm> (twice '(+ 1 2))
(begin (+ 1 2) (+ 1 2))
```

Wouldn't it be nice to achieve the same effect without needing to quote?

Macros (2 of 9)

```
scm> (define (twice expr) (list 'begin expr expr))
twice
scm> (eval (twice '(print 2)))
2
2

scm> (define-macro (twice expr) (list 'begin expr expr))
twice
scm> (twice (print 2))
2
2
```

Macros (3 of 9)

- A **macro** is an operation performed on the source code of a program before evaluation (exists in many languages)
- Scheme has a `define-macro` special form, which allows us to have control over which expression(s) get evaluated and when
  - Lets us define our own special forms!

Evaluation procedure of a macro call expression:

1. Evaluate the operator subexpression, which evaluates to a macro (not a normal procedure)
2. Call the macro procedure on the operand expressions **without evaluating them first** (this is why we got equivalent behavior using quotes)
3. Evaluate the expression returned from the macro procedure

Macros (4 of 9)

```
(define-macro (check expr)
    (list 'if expr ''passed ''failed)
)
(define x -2)
scm> (check (> x 0))
failed
```

Step 1: When evaluating a macro call expression, first we evaluate the operator, which evaluates to a macro procedure

Macros (5 of 9)

```
(define-macro (check expr)
    (list 'if expr ''passed ''failed)
)
(define x -2)
scm> (check (> x 0))
failed
```

Step 2: The arguments to the macro are passed into its body **without being evaluated** (can think of this as the "substitution" step)

Macros (6 of 9)

```
(define-macro (check expr)
    (list 'if (> x 0) ''passed ''failed)
)
(define x -2)
scm> (check (> x 0))
failed
```

Step 2: The arguments to the macro are passed into its body **without being evaluated** (can think of this as the "substitution" step)

Macros (7 of 9)

```
; 3a: macro body after substitution:
(list 'if (> x 0) ''passed ''failed)
; 3b: lookup value of x in the environment:
(list 'if (> -2 0) ''passed ''failed)
; 3c: apply list procedure:
(if #f 'passed 'failed)
; 3d: evaluate if special form:
'failed
; 3e: value displayed in the interpreter as:
failed
```

Step 3: The body of the macro is evaluated

Macros (8 of 9)

What if we wanted to change the check macro so that it also displays the expression if it failed the check?

```
(define-macro (check expr)
    (list 'if expr ''passed ''failed)
)

(define-macro (check expr)
    (list 'if expr ''passed
        (list 'quote (list 'failed: expr))
    )
)
```

Macros (9 of 9)

Can we make the macro even more readable and shorter? Yes, with quasiquote and unquote! (Eliminates step 3c from slide 35)

```
(define-macro (check expr)
    (list 'if expr ''passed
        (list 'quote (list 'failed: expr))
    )
)


(define-macro (check expr)
    `(if ,expr 'passed
        '(failed: ,expr)
    )
)
```

5 min break

Practice

(reminder to self to use high contrast theme)

30 min

- Reverse (Tail Recursive)
- Map (Tail Recursive)
- For Macro

Reverse (Tail Recursive)

- Implement `reverse` so that it is tail recursive
- Download starter code `21.scm` from the course website under Lecture 21
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Map (Tail Recursive)

- Implement `map` so that it is tail recursive
  - Hint: How can you use `reverse` to implement `map`?
- Download starter code `21.scm` from the course website under Lecture 21
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

For Macro

- Implement the `for` macro
  - **Hint:** How can you use `map` to implement `for`?
  - **Challenge:** Implement this 2 ways, one ***without*** quasiquotes and unquote, and one ***with*** them
- Download starter code `21.scm` from the course website under Lecture 21
- Run tests by copying and pasting your code into a new file in code.cs61a.org and clicking on the red test tube button on the top right
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?