# TAIL RECURSION & INTERPRETERS Solutions

COMPUTER SCIENCE MENTORS 61A

November 7–November 11, 2022

## 1 Tail Recursion

Recursion has an efficiency problem. Consider the factorial function. In order to calculate `factorial(6)`, we have to call `factorial(5)`, `factorial(4)`, ..., `factorial(0)`. In all, 7 frames are opened to calculate this one value. Now what if we tried `factorial(1000000)`? 1,000,001 frames would be opened, and our computer would surely crash.

There must be a better way. In languages such as Python, baked-in iterative tools like `for` and `while` loops allow us to complete large repetitive tasks with a small amount of computer resources. In Scheme, which lacks native support for iteration, are we doomed to inefficiency?

No. Because Scheme implements a feature called **tail recursion optimization**, certain kinds of recursive functions can be computed in constant space, just like a Python `while` loop. These "tail-recursive" functions are simply recursive functions where the *very last* thing we do during execution is return the recursive call:

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

```
(define (fact-tail n)
  (define (fact-help product n)
    (if (= n 0)
      product
      (fact-help (* n product) (- n
          1))))
  (fact-help 1 n))
```

The implementation of `fact-tail` on the right is tail-recursive; when we make a recursive call, it is the last thing we do in the function's execution. The implementation on the left is not tail recursive; after the recursive call to `fact` returns, we have to still multiply it by `n`. That multiplication, not the recursive call, is the last thing we do in the function's execution.

Let's break `fact` down some more. In order to determine the value of `fact(6)`, we have to pause and save the current frame to calculate the value of `fact(5)` and then resume execution to multiply by 6 and get our final answer. However, if we define a tail-recursive function in which no further calculations are done after the recursive call, none of the values in the current frame have to be saved. So we can close the current as we make the next recursive call, ensuring that we only have one frame open at any time. This is the mechanism behind tail recursion.

Created by Gabe Classon, Aditya Balasubramanian, Alyssa Smith, Esther Shen, Maya Romero, and Manas Khatore

Generally, it's best to think about tail recursion in this conceptual manner. However, there are formal rules for determining whether recursive calls can be optimized:

We can optimize recursive calls that are in a tail context. In Python, which does not optimize tail calls, the tail contexts are the return statements. In Scheme, a tail context is recursively defined as the last line (return value) of a function or

- the second or third operand in a tail context **`if`** expression
- any of the non-predicate sub-expressions in a tail context **`cond`** expression (i.e. the second expression of each clause)
- the last operand in a tail context **`and`** or **`or`** expression
- the last operand in a tail context **`begin`** expression's body
- the last operand in a tail context **`let`** expression's body

As hinted at in the factorial example, the general way to convert a recursive function to a tail recursive one is to move the calculation from outside the recursive call to one of the recursive call arguments and thereby accumulate the result. This frequently requires the creation of a tail-recursive helper.

1. Consider the following function:

```
(define (sum-list lst)
  (if (null? lst)
    0
    (+ (car lst) (sum-list (cdr lst)))
  )
)
```

   (a) What are all of expressions of `sum-list` that are in tail contexts? (Hint: there are three.) Is the call to `sum-list` tail recursive?

   The tail context expressions are:

   - The entire **`if`** expression is in a tail context because it is the last operand of the body of a function.
   - The `0` is in a tail context because it is the second operand of a tail-context **`if`** expression.
   - The expression `(+ (car lst)(sum-list (cdr lst)))` is in a tail context because it is the third operand of a tail-context **`if`** expression.

   The call to `sum-list` is not tail recursive because it is not in a tail context. On a more conceptual level, it is not the last expression we evaluate; after the recursive call returns, we still have to perform the addition operation.

(b) As we increase the length of `lst`, how does the total amount of space used by `sum-list` change? Why?

Space usage increases linearly with the length of `lst`. The recursive call to `sum-list` is not in a tail context, so Scheme is not able to optimize it. That means that each time `sum-list` is recursively called, another active frame is opened, taking up more space.

(c) Rewrite `sum-list` to be tail recursive.

```
(define (sum-list-tail lst)









)
```

```
(define (sum-list-tail lst)
  (define (sum-list-helper lst sofar)
    (if (null? lst)
      sofar
      (sum-list-helper (cdr lst) (+ sofar (car lst)))
    )
  )
  (sum-list-helper lst 0)
)
```

(d) As we increase the length of `lst`, how does the total amount of space used by our optimized version of `sum-list` change? Why?

Space usage is constant due to tail call optimization.

2. Implement `filter-lst`, which takes in a one-argument function `f` and a list `lst`, and returns a new list containing only the elements in `lst` for which `f` returns true. Your function must use a constant number of active frames.

   *Hint: recall that the built-in* `append` *procedure concatenates two lists together.*

```
;Doctests
scm> (filter-lst (lambda (x) (> x 2)) '(1 2 3 4 5))
(3 4 5)
```

```
(define (filter-lst f lst)

)
```

```
(define (filter-lst f lst)
  (define (filter-tail lst so-far)
     (cond ((null? lst) so-far)
           ((f (car lst)) (filter-tail (cdr lst)
                                       (append so-far (list (car lst)))))
           (else (filter-tail (cdr lst) so-far))))
  (filter-tail lst nil))
```

3. Slice and Dice

(a) Implement `slice`, which takes in a list `lst`, a starting index `i`, and an ending index `j`, and returns a new list containing the elements of `lst` from index `i` to `j - 1`.

```
;Doctests
scm> (slice '(0 1 2 3 4) 1 3)
(1 2)
scm> (slice '(0 1 2 3 4) 3 5)
(3 4)
scm> (slice '(0 1 2 3 4) 3 1)
()

(define (slice lst i j)

)
```

```
(define (slice lst i j)
    (cond ((or (null? lst) (>= i j)) nil)
          ((= i 0) (cons (car lst) (slice (cdr lst) i (- j 1))))
          (else (slice (cdr lst) (- i 1) (- j 1)))))
```

(b) Now implement `slice` tail recursively!

```
(define (slice lst i j)

)
```

```scheme
(define (slice lst i j)
  (define (slice-tail lst i j lst-so-far)
      (cond ((or (null? lst) (>= i j)) lst-so-far)
            ((= i 0) (slice-tail (cdr lst) i (- j 1) (append
               lst-so-far (list (car lst)))))
            (else (slice-tail (cdr lst) (- i 1) (- j 1)
               lst-so-far))))
  (slice-tail lst i j nil))
```

Alternate Solution:

```scheme
(define (slice lst i j)
  (define (slice-tail lst index lst-so-far)
      (cond ((or (null? lst) (= index j)) lst-so-far)
            ((<= i index) (slice-tail (cdr lst) (+ index 1) (append
               lst-so-far (list (car lst)))))
            (else (slice-tail (cdr lst) (+ index 1) lst-so-far))))
  (if (< i j) (slice-tail lst 0 nil) nil))
```

## 2 Interpreters

An **interpreter** is a computer program that understands, processes, and executes other programs. The Scheme interpreter we will cover in CS 61A is built around the **Read-Eval-Print Loop**, which consists of the following steps:

1. **Read** the raw input and parse it into a data structure we can easily handle.
2. **Evaluate** the parsed expression.
3. **Print** the result to output.

One of the challenges of designing interpreters is to represent the input in a way that the interpreter's language can understand. For example, since our Scheme interpreter is written in Python, we need to parse Scheme tokens into a usable Python representation. Conveniently, every Scheme call expression and special form is represented in Scheme as a linked list. Therefore, we will represent Scheme lists with the `Pair` class, which is a type of linked list.

Just like `Link`, a `Pair` instance has two attributes, `first` and `second`, which contain the first element and rest of the linked list, respectively. Instead of using `Link.empty` to represent an empty list, `Pair` uses `nil`.

For example, during the read step, the Scheme expression `(+ 1 2)` would be **tokenized** into `'(', '+', '1', '2', ')'` and then organized into a `Pair` instance as `(Pair('+', Pair(1, Pair(2, nil)))).`

Once we have parsed our input, we evaluate the expression by calling `scheme_eval` on it. If it's a procedure call, we recursively call `scheme_eval` on the operator and the operands. Then we return the result of calling `scheme_apply` on the evaluated operator and operands, which computes the procedure call. If it's a special form, the relevant evaluation rules are followed in a similar matter.

For example, when we provide `(+ 1 (+ 2 3))` as input to the interpreter, the following happen:

- `(+ 1 (+ 2 3))` is parsed to `Pair('+', Pair(1, Pair(Pair('+', Pair(2, Pair(3, nil))), nil)))`

- The interpreter recognizes this is a procedure call.
- `scheme_eval` is called on the operator, `'+'`, and returns the addition procedure.
- `scheme_eval` is called on the operand `1` and returns `1`.
- `scheme_eval` is called on the operand `Pair('+', Pair(2, Pair(3, nil)))`.
  - The interpreter recognizes this is a procedure call.
  - `scheme_eval` is called on the operator, `'+'`, and returns the addition procedure.
  - `scheme_eval` is called on the operand `2` and returns `2`.
  - `scheme_eval` is called on the operand `3` and returns `3`.
  - `scheme_apply` is called on the evaluated procedure and parameters (`Pair(2, Pair(3, nil))`) and returns `5`.
- `scheme_apply` is called on the evaluated procedure and parameters (`Pair(1, Pair(5, nil))`) and returns `6`.
- `6` is printed to output.

1. The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project.

   (a) What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?

   The read stage returns a representation of the code that is easier to process later in the interpreter by putting it in a new data structure. In our interpreter, it takes in a string of code, and outputs a Pair representing an expression (which is really just the same as a Scheme list).

   (b) What are the two components of the read stage? What do they do?

   The read stage consists of

   1. The lexer, which breaks the input string and breaks it up into tokens (individual characters or symbols)
   2. The parser, which takes that string of tokens and puts it into the data structure that the read stage outputs (in our case, a Pair).

   (c) Write out the constructor for the `Pair` object that the read stage creates from the input string `(define (foo x) (+ x 1))`

   Pair("define", Pair(Pair("foo", Pair("x", nil)), Pair(Pair("+", Pair("x", Pair(1, nil))), nil)))

   (d) For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

   We could check to see that it's a define special form by checking if `p.first == "define"`.

   We could get its name by accessing `p.second.first.first` and get the body of the function with `p.second.second.first`.

2. Circle or write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

| | | | | |
|---|---|---|---|---|
| `scheme_eval` | 1 | 3 | 4 | 6 |
| `scheme_apply` | 1 | 2 | 3 | 4 |

6 `scheme_eval`, 1 `scheme_apply`. Evals: (1) on the entire expression, (2) on `1` (**`if`** is not evaluated), (3) on (+ 2 3), (4-6) on +, 2, 3. Apply: (1) with applying + on (+ 2 3).

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

| | | | | |
|---|---|---|---|---|
| `scheme_eval` | 6 | 8 | 9 | 10 |
| `scheme_apply` | 1 | 2 | 3 | 4 |

8 `scheme_eval`, 1 `scheme_apply`.

```
(define (square x) (* x x))

(+ (square 3) (- 3 2))
```

| | | | | |
|---|---|---|---|---|
| `scheme_eval` | 2 | 5 | 14 | 24 |
| `scheme_apply` | 1 | 2 | 3 | 4 |

14 `scheme_eval`, 4 `scheme_apply`.

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```

13 `scheme_eval`, 3 `scheme_apply`.

3. Identify the number of calls to `scheme_eval` and the number of calls to `scheme_apply`.

(a)
```
scm> (define pi 3.14)
pi
scm> (define (hack x)
          (cond
            ((= x pi) 'pwned)
            ((< x 0) (hack pi))
            (else (hack (- x 1)))))
hack
```

3 `scheme_eval`, 0 `scheme_apply`
Evals: (1) on the first line, (2) on 3.14, and (3) on the second line.

(b)
```
scm> (hack 3.14)
pwned
```

9 `scheme_eval`, 2 `scheme_apply`
Evals: (1) The entire expression, (2) hack, (3) 3.14, (4) hack's entire cond expression, (5) (= x pi), (6-8) =/x/pi, (9) 'pwned
Apply: (1) hack in (hack 3.14), (2) = in (= x pi)

(c)
```
scm> ((lambda (x) (hack x)) 0)
pwned
```

39 `scheme_eval`, 10 `scheme_apply`

## 3 Scheme Challenge

1. Finish the functions `max` and `max-depth`. `max` takes in two numbers and returns the larger. Function `max-depth` takes in a list `lst` and returns the maximum depth of the list. In a nested scheme list, we define the depth as the number of scheme lists a sublist is nested within. A scheme list with no nested lists has a `max-depth` of 0.

```
;doctests
scm> (max 1 5)
5
scm> (max-depth '(1 2 3))
0
scm> (max-depth '(1 2 (3 (4) 5)))
2
scm> (max-depth '(0 (1 (2 (3 (4) 5) 6) 7))
4

(define (max x y) ____________________________________)

(define (max-depth lst)
    (define (helper lst curr)
        (cond
            ((__________) ________________________)

            ((__________) (max ______________________________
```

```
                                           ________________________________))

            (else (helper ________________________))
        )
    )
    (____________________________)
)
```

```
(define (max x y) (if (> x y) x y))

(define (max-depth lst)
    (define (helper lst curr)
            (cond
              ((null? lst) curr)
              ((pair? (car lst)) (max (helper (car lst)
                                              (+ 1 curr))
                                      (helper (cdr lst) curr)))
              (else (helper (cdr lst) curr))
            )
      )
    (helper lst 0)
)
```