CS 61A
Summer 2025

Tail Calls
Discussion 10: July 31, 2025

Tail Calls

When writing a recursive procedure, it's possible to write it in a **tail recursive** way, where all of the recursive calls are tail calls. A **tail call** occurs when a function calls another function as the last action of the current frame.

Consider this implementation of `factorial` that is *not* tail recursive:

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

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(factorial (- n 1))`, the function still needs to multiply that result with `n`. The final expression that is evaluated is a call to the multiplication function, not `factorial` itself. Therefore, the recursive call is `not` a tail call.

Here's a visualization of the recursive process for computing `(factorial 6)` :

```
(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
```

The interpreter first must reach the base case and only then can it begin to calculate the products in each of the earlier frames.

We can rewrite this function using a helper function that remembers the temporary product that we have calculated so far in each recursive step.

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

`fact-tail` makes a single recursive call to `fact-tail`, and that recursive call is the last expression to be evaluated, so it is a tail call. Therefore, `fact-tail` is a tail recursive process.

Here's a visualization of the tail recursive process for computing `(factorial 6)`:

```
(factorial 6)
(fact-tail 6 1)
(fact-tail 5 6)
(fact-tail 4 30)
(fact-tail 3 120)
(fact-tail 2 360)
(fact-tail 1 720)
(fact-tail 0 720)
720
```

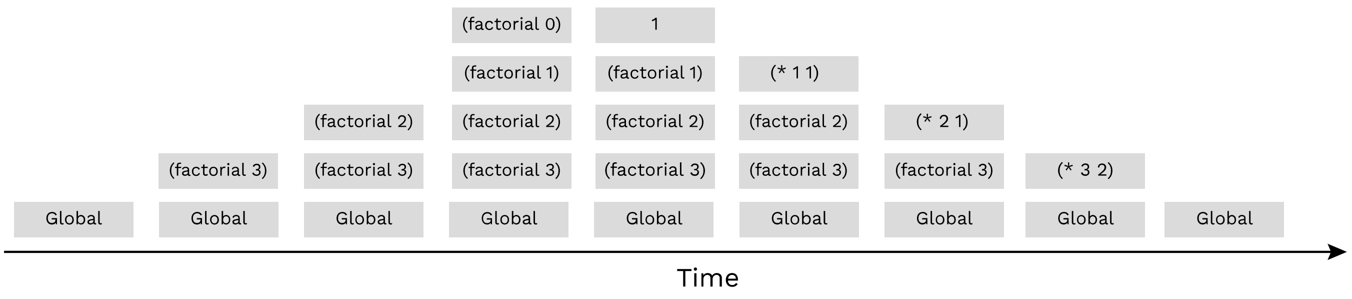
The interpreter needed less steps to come up with the result, and it didn't need to re-visit the earlier frames to come up with the final product.

In this example, we've utilized a common strategy in implementing tail-recursive procedures which is to pass the result that we're building (e.g. a list, count, sum, product, etc.) as a argument to our procedure that gets changed across recursive calls. By doing this, we do not have to do any computation to build up the result after the recursive call in the current frame, instead any computation is done *before* the recursive call and the result is passed to the next frame to be modified further. Often, we do not have a parameter in our procedure that can store this result, but in these cases we can define a helper procedure with an extra parameter(s) and recurse on the helper. This is what we did in the `factorial` procedure above, with `fact-tail` having the extra parameter `result`.

Tail Call Optimization

When a recursive procedure is not written in a tail recursive way, the interpreter must have enough memory to store all of the previous recursive calls.

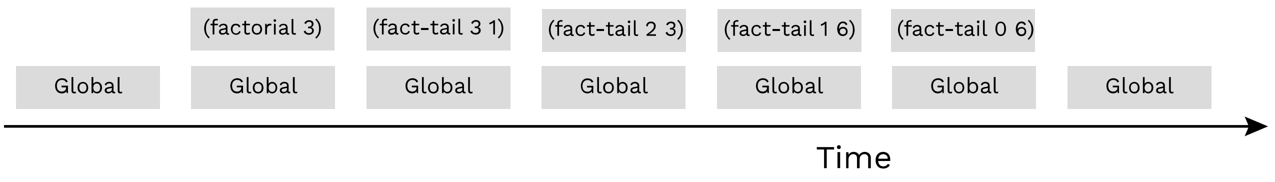
For example, a call to the `(factorial 3)` in the non tail-recursive version must keep the frames for all the numbers from 3 down to the base case, until it's finally able to calculate the intermediate products and forget those frames:



For non tail-recursive procedures, the number of active frames grows proportionally to the number of recursive calls. That may be fine for small inputs, but imagine calling `factorial` on a large number like 10000. The interpreter would need enough memory for all 1000 calls!

Fortunately, proper Scheme interpreters implement **tail-call optimization** as a requirement of the language specification. TCO ensures that tail recursive procedures can execute with a constant number of active frames, so programmers can call them on large inputs without fear of exceeding the available memory.

When the tail recursive `factorial` is run in an interpreter with tail-call optimization, the interpreter knows that it does not need to keep the previous frames around, so it never needs to store the whole stack of frames in memory:



Tail-call optimization can be implemented in a few ways:

1. Instead of creating a new frame, the interpreter can just update the values of the relevant variables in the current frame (like `n` and `result` for the `fact-tail` procedure). It reuses the same frame for the entire calculation, constantly changing the bindings to match the next set of parameters.
2. How our 61A Scheme interpreter works: The interpreter builds a new frame as usual, but then *replaces* the current frame with the new one. The old frame is still around, but the interpreter no longer has any way to get to it. When that happens, the Python interpreter does something clever: it *recycles* the old frame so that the next time a new frame is needed, the system simply allocates it out of recycled space. The technical term is that the old frame becomes "garbage", which the system "garbage collects" behind the programmer's back.

Tail Context

When trying to identify whether a given function call within the body of a function is a tail call, we look for whether the call expression is in **tail context**.

Given that each of the following expressions is the last expression in the body of the function, the following expressions are tail contexts:

1. the second or third operand in an `if` expression
2. any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
3. the last operand in an `and` or an `or` expression
4. the last operand in a `begin` expression's body
5. the last operand in a `let` expression's body

For example, in the expression `(begin (+ 2 3) (- 2 3) (* 2 3))`, `(* 2 3)` is a tail call because it is the last operand expression to be evaluated.

Q1: Is Tail Call

For each of the following procedures, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of active frames.

```
(define (question-a x)
  (if (= x 0) 0
      (+ x (question-a (- x 1)))))
```

```
(define (question-b x y)
  (if (= x 0) y
      (question-b (- x 1) (+ y x))))
```

```
(define (question-c x y)
  (if (> x y)
      (question-c (- y 1) x)
      (question-c (+ x 10) y)))
```

```
(define (question-d n)
  (if (question-d n)
      (question-d (- n 1))
      (question-d (+ n 10))))
```

```
(define (question-e n)
  (cond ((<= n 1) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

Q2: Sum

Write a tail recursive function that takes in a Scheme list and returns the numerical sum of all values in the list. You can assume that the list contains only numbers (no nested lists).

```
scm> (sum '(1 2 3))
6
scm> (sum '(10 -3 4))
11
```

```
(define (sum lst)
  'YOUR-CODE-HERE
)

(expect (sum '(1 2 3)) 6)
(expect (sum '(10 -3 4)) 11)
```

Q3: Reverse

Write a tail-recursive function `reverse` that takes in a Scheme list a returns a reversed copy.

```
scm> (reverse '(1 2 3))
(3 2 1)
scm> (reverse '(0 9 1 2))
(2 1 9 0)
```

```
(define (reverse lst)
  'YOUR-CODE-HERE









)

(expect (reverse '(1 2 3)) (3 2 1))
(expect (reverse '(0 9 1 2)) (2 1 9 0))
```

Exam Practice

Q4: Num Splits

Given a list of numbers `s` and a target difference `d`, write a function `num_splits` that calculates how many different ways are there to split `s` into two subsets, such that the sum of the first is within `d` of the sum of the second. The number of elements in each subset can differ.

You may assume that the elements in `s` are distinct and that `d` is always non-negative.

Note that the order of the elements within each subset does not matter, nor does the order of the subsets themselves. For example, given the list `[1, 2, 3]`, you should not count `[1, 2], [3]` and `[3], [1, 2]` as distinct splits.

```python
def num_splits(s, d):
    """Return the number of ways in which s can be partitioned into two
    sublists that have sums within d of each other.

    >>> num_splits([1, 5, 4], 0)  # splits to [1, 4] and [5]
    1
    >>> num_splits([6, 1, 3], 1)  # no split possible
    0
    >>> num_splits([-2, 1, 3], 2) # [-2, 3], [1] and [-2, 1, 3], []
    2
    >>> num_splits([1, 4, 6, 8, 2, 9, 5], 3)
    12
    """
    "*** YOUR CODE HERE ***"
```

Q5: Cycles

The `Link` class can represent lists with cycles. That is, a list may contain itself as a sublist.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.rest.rest.rest = s
>>> s.rest.rest.rest.rest.rest.first
3
```

Implement `has_cycle`,that returns whether its argument, a `Link` instance, contains a cycle.

```
def has_cycle(link):
    """Return whether link contains a cycle.

    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle(s)
    True
    >>> t = Link(1, Link(2, Link(3)))
    >>> has_cycle(t)
    False
    >>> u = Link(2, Link(2, Link(2)))
    >>> has_cycle(u)
    False
    """
    "*** YOUR CODE HERE ***"
```

Extra challenge (Optional): Implement `has_cycle` without keeping track of all `Link` objects you've already seen. The solution is short (less than 20 lines of code), but requires a clever idea. Try to discover the solution yourself before asking around.

```
def has_cycle_constant(link):
    """Return whether link contains a cycle.

    >>> s = Link(1, Link(2, Link(3)))
    >>> s.rest.rest.rest = s
    >>> has_cycle_constant(s)
    True
    >>> t = Link(1, Link(2, Link(3)))
    >>> has_cycle_constant(t)
    False
    """
    "*** YOUR CODE HERE ***"
```

Q6: Checking it Twice

Draw an environment diagram for the following program. Then, check your work by stepping through the diagram with PythonTutor.

> Some things to remember: * When you mutate a list, you are changing the original list. * When you concatenate two lists, you are creating a new list. * When you assign a name to an existing object, you are creating another reference to that object rather than creating a copy of that object.

```
def push(t, x):
    s = list(t)
    s[1].append(x)

def pull():
    s = list(t)
    u = s.pop()
    return u[1][0]

s = [7, [8, 9]]
t = [6, s]

push(s, pull())
print(t)
```

Q7: In-order Traversal

Write a function that returns a generator that generates an "in-order" traversal, in which we yield the value of every node in order from left to right, assuming that each node has either 0 or 2 branches.

```
def in_order_traversal(t):
    """
    Generator function that generates an "in-order" traversal, in which we
    yield the value of every node in order from left to right, assuming that each node
    has either 0 or 2 branches.

    For example, take the following tree t:
            1
        2       3
    4     5
         6  7

    We have the in-order-traversal 4, 2, 6, 5, 7, 1, 3

    >>> t = Tree(1, [Tree(2, [Tree(4), Tree(5, [Tree(6), Tree(7)])]), Tree(3)])
    >>> list(in_order_traversal(t))
    [4, 2, 6, 5, 7, 1, 3]
    """
    "*** YOUR CODE HERE ***"
```

Challenge

Q8: After Party

Implement after, which takes a linked list s and values a and b. It returns whether an element of s equal to b appears after an element of s equal to a.

```
def after(s, a, b):
    """Return whether b comes after a in linked list s.
    >>> t = Link(3, Link(6, Link(5, Link(4))))
    >>> after(t, 6, 4)
    True
    >>> after(t, 4, 6)
    False
    >>> after(t, 6, 6)
    False
    """
    "*** YOUR CODE HERE ***"
```

Submit Attendance

You're done! Excellent work this week. Please be sure to ask your section TA for the attendance form link and fill it out for credit. (one submission per person per section).