

1 Streams

In Python, we can use iterators to represent infinite sequences (for example, the generator for all natural numbers). However, Scheme does not support iterators. Let's see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

```
scm> (define (naturals n)
      (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because `cons` is a regular procedure and both its operands must be evaluated before the pair is constructed, we cannot create an infinite sequence of integers using a Scheme list.

Instead, our Scheme interpreter supports *streams*, which are *lazy* Scheme lists. The first element is represented explicitly, but the rest of the stream's elements are computed only when needed. Computing a value only when it's needed is also known as *lazy evaluation*.

```
scm> (define (naturals n)
      (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream:

```
(cons-stream <operand1> <operand2>)
```

`cons-stream` is a special form because the second operand is not evaluated when evaluating the expression. To evaluate this expression, Scheme does the following:

1. Evaluate the first operand.
2. Construct a promise containing the second operand.
3. Return a pair containing the value of the first operand and the promise.

To actually get the rest of the stream, we must call `cdr-stream` on it to force the promise to be evaluated. Note that this argument is only evaluated once and is then stored in the promise; subsequent calls to `cdr-stream` returns the value without recomputing it. This allows us to efficiently work with infinite streams like the `naturals` example above. We can see this in action by using a non-pure function to compute the rest of the stream:

```
scm> (define (compute-rest n)
...>   (print 'evaluating!)
...>   (cons-stream n nil))
compute-rest
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

Here, the expression `compute-rest 1` is only evaluated the first time `cons-stream` is called, so the symbol `evaluating!` is only printed the first time.

Streams are very similar to Scheme lists in that they are also recursive structures. Just like the `cdr` of a Scheme list is either another Scheme list or `nil`, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that whereas both arguments to `cons` are evaluated upon calling `cons`, the second argument to `cons-stream` isn't evaluated until the first time that `cdr-stream` is called.

Here's a summary of what we just went over:

- `nil` is the empty stream
- `cons-stream` constructs a stream containing the value of the first operand and a promise to evaluate the second operand
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

[Video walkthrough](#)

Questions

1.1 What would Scheme display?

As you work through these problems, remember that streams have two important components:

- Lazy evaluation – so the remainder of the stream isn’t computed until explicitly requested.
- Memoization – so anything we compute won’t be recomputed.

The examples here stretch these concepts to the limit. In most practical use cases, you may find you rarely need to redefine functions that compute the remainder of the stream.

```
scm> (define (has-even? s)
      (cond ((null? s) #f)
            ((even? (car s)) #t)
            (else (has-even? (cdr-stream s)))))

has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums

scm> nums

(1 . #[promise (not forced)])

scm> (cdr nums)

#[promise (not forced)]

scm> (cdr-stream nums)

(9 . #[promise (not forced)])

scm> nums

(1 . #[promise (forced)])

scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)

(9 . #[promise (not forced)])

scm> (cdr-stream (cdr-stream nums))

(10 . #[promise (not forced)])

scm> (has-even? nums)

True
```

[Video walkthrough](#)

1.2 Write a function range-stream which takes a start and end, and returns a stream

that represents the integers between `start` and `end - 1` (inclusive).

```
(define (range-stream start end)
```

```
  (if (_____)
```

```
      nil
```

```
      (cons-stream _____)))
```

```
  (if (= start end)
```

```
      nil
```

```
      (cons-stream start (range-stream (+ start 1) end))))
```

It might help to compare this to the version of `range` for regular (non-stream)

Scheme lists:

```
(define (range start end)
```

```
  (if (= start end)
```

```
      nil
```

```
      (cons start (range (+ start 1) end))))
```

```
scm> (define s (range-stream 1 5))
```

```
s
```

```
scm> (car (cdr-stream s))
```

```
2
```

- 1.3 Write a function `slice` which takes in a stream `s`, a `start`, and an `end`. It should return a Scheme list that contains the elements of `s` between index `start` and `end`, not including `end`. If the stream ends before `end`, you can return `nil`.

```
(define (slice s start end)

  (cond
    ((or (null? s) (= end 0)) nil)
    (> start 0)
    (slice (cdr-stream s) (- start 1) (- end 1)))
  (else
   (cons (car s)
         (slice (cdr-stream s) (- start 1) (- end 1))))))
```

```
scm> (define nat (naturals 0)) ; See naturals procedure on page 1
nat
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

- 1.4 Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! Use it to define a few of our favorite sequences. We've defined the function `combine-with` for you below, as well as an example of how to use it to define the stream of even numbers.

```
(define (combine-with f xs ys)
  (if (or (null? xs) (null? ys))
      nil
      (cons-stream
        (f (car xs) (car ys))
        (combine-with f (cdr-stream xs) (cdr-stream ys)))))

scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

For these questions, you may use the `naturals` stream in addition to `combine-with`.

- i. (define factorials

```
  (cons-stream 1 (combine-with * (naturals 1) factorials)))

scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

(Continued on next page)

ii. **(define fibs**

```
(cons-stream 0
  (cons-stream 1
    (combine-with + fibs (cdr-stream fibs)))))
```

```
scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```

iii. Write **exp**, which returns a stream where the n th term represents the degree- n polynomial expansion for e^x , which is $\sum_{i=0}^n x^i/i!$.

You may use **factorials** in addition to **combine-with** and **naturals** in your solution.

(define (exp x)

```
(let ((terms (combine-with (lambda (a b) (/ (expt x a) b))
                           (cdr-stream (naturals 0))
                           (cdr-stream factorials))))
  (cons-stream 1 (combine-with + terms (exp x)))))
```

```
scm> (slice (exp 2) 0 5)
(1 3 5 6.333333333 7 7.266666667)
```

2 Tail Recursion

Scheme implements tail-call optimization, which allows programmers to write recursive functions that use a constant amount of space. A **tail call** occurs when a function calls another function as its **last action of the current frame**. In this case, the frame is no longer needed, and we can remove it from memory. In other words, if this is the last thing you are going to do in a function call, we can reuse the current frame instead of making a new frame.

Consider this implementation of `factorial`.

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

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(fact (- 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 `fact` itself. Therefore, the recursive call is **not** a tail call.

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 (fact 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. We say that a recursive function is **tail recursive** if all of its recursive calls are tail calls.

Using a constant number of frames

Tail recursive processes can use a constant amount of memory because each recursive call frame does not need to be saved.

Our original implementation of `fact` required the program to keep each frame open because the last expression multiplies the recursive result with `n`. Therefore, at each frame, we need to remember the current value of `n`.

In contrast, the tail recursive `fact-tail` does not require the interpreter to remember the values for `n` or `result` in each frame. Instead, we can just *update* the value of `n` and `result` of the current frame! Therefore, we can keep reusing a single frame to complete this calculation.

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, we consider the tail context of each expression to be:

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

[Video walkthrough](#)

Questions

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

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

In the recursive case, the last expression that is evaluated is a call to `+`. Therefore, the recursive call is not in tail context, and each of the frames remain active. This function uses $\Theta(n)$ frames.

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

The recursive call is the third operand in the **if** expression, so it is in tail context. This means that the last expression that will be evaluated in the body of this function is the recursive function call, so this function uses $\Theta(1)$ frames.

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

The recursive calls are the second and third operands of the **if** expression. Only one of these calls is actually evaluated, and whichever one it is will be the last

expression evaluated in the body of the function. This function therefore uses $\Theta(1)$ frames.

Note that if you actually try and evaluate this function, it will never terminate. But at least it won't crash from hitting max recursion depth!

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

The second and third recursive calls are in tail context, but the first is not. Since not all the recursive calls are tail calls, this function does not use a constant number of frames. [Video walkthrough](#)

```
(define (question-e n)
  (cond ((= n 0) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3)))))))
```

The second and third recursive calls are the second expressions in a clause, so they are in tail context. However, the first recursive call is not in tail context. Since not all recursive calls are tail calls, this function is not tail recursive and does not use a constant number of frames.

Writing tail recursive functions

Before we jump into questions, a quick tip for defining tail recursive functions is to use helper functions. This is so that we can keep track of extra parameters and make all recursive calls in tail context. Examples of such parameters include something like `total`, `counter`, or `result`, like we saw in `fact-tail`.

- 2.2 Write a tail recursive function that returns the n th fibonacci number. We define $\text{fib}(0) = 0$ and $\text{fib}(1) = 1$.

```
(define (fib n)
  (define (fib-sofar _____)

    (if _____

        _____

        (fib-sofar _____)

        (fib-sofar _____)))

(define (fib n)
  (define (fib-sofar i prev curr)
    (if (= i n)
```

```

      prev
      (fib-sofar (+ i 1) curr (+ prev curr)))
(fib-sofar 0 0 1))

```

Video walkthrough

- 2.3 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 is well-formed and contains only numbers (no nested lists).

```
(define (sum lst)
```

```

(define (sum lst)
  (define (sum-sofar lst current-sum)
    (if (null? lst)
        current-sum
        (sum-sofar (cdr lst) (+ (car lst) current-sum))))
  (sum-sofar lst 0))

```

Video walkthrough

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

```
(define (reverse lst)
```

```

(define (reverse lst)
  (define (reverse-sofar lst lst-sofar)
    (if (null? lst)
        lst-sofar
        (reverse-sofar (cdr lst) (cons (car lst) lst-sofar))))
  (reverse-sofar lst nil))

```