Infinite Prime Sieve in Lisp

Leon Rische

[2017-09-01 Fri 15:48]

Contents

1 Setup

The code examples are written in a lisp I'm working on.

To try it out, clone the git repo, make sure rust is installed and run `cargo run repl` to get an interactive REPL or `cargo run run example.scm` to evaluate a file.

2 Basic Syntax

- Booleans: `true` and `false`

- The empty List: `nil` or `'()` or `(list)` (they are pretty much equivalent)
- `(def a 1)` defines a variable named `a` with a value of `1`.
- `(set! a 2)` overwrites `a` with `2`
- `(fn (n div) (` 0 (% n div))= creates an anonymous function = with two arguments `n` and `div` that checks if `n` is divisible by `div`.
- `(defn name (arg1 arg2) body)` is syntax sugar for `(def name (fn (arg1 arg2) body))` and can be used to define named functions
- `(cons 1 2)` creates a **pair** of two values
- `(if test consequent alternative)` is equivalent to `if (test) { consequent } else {alternative }` in other languages
- `(do expr1 expr2 ...)` evaluates each of the arguments and returns the result of the last one. It can be used to do more than one thing inside the `consequent` of an `if` etc.
- `(inc n)` and `(dec n)` are equiv. to `(+ n 1)` and `(- n 1)`
- `fst` and `rst` ("first" and "rest") are equivalent to `car` and `cdr` in languages like Chicken Scheme and can be used to access the first and second element of a `cons` pair
- `print` and `println` print values on the screen, without or with a newline

3 Special Forms

Given an expression like `(println (+ 1 2))` the 'normal' way (in a language that is not lazily evaluated) to treat it would be to evaluate all elements (`println` to a function reference, `(+ 1 2)` to `3`) and then apply the rest of the elements to the first.

This is good enough most times, but consider this example:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn unless (test consequent alternative) (if test alternative consequent))

Looks pretty harmless, but what happens if we try to use it like this?

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def n 5) (unless (= 0 ((println "n is odd") (println "n is even"))

The result will be something like

```
n is odd
n is even
=> undefined (ignored by the repl)
```

With the 'normal' way of evaluating expressions, the first step would be to evaluate `(= 0 (% n 2))` to `true`, `(println "n is odd")` to `undefined` (because it does not return anything useful) and `(println "n is even")` to `undefined`, too.

The remaining expression is `(if true undefined undefined)` and that evaluates to `undefined`.

To avoid this, `unless` (and some other syntax constructs like `if`, `delay`) must be implemented as **special forms**.

Simplified this means they are called with 'raw', unevaluated arguments and can choose when (and if) to evaluate them themselves.

An implementation of `unless` as special form would need to evaluate the test first and then, depending on the result, evaluate either the consequent or the alternative.

4 Promises

`delay` and `force` can be used to delay the evaluation of an expression.

A naive implementation would be to `(defn delay (body) (fn () body))` (just wrap an anonymous function around the body) and `(defn force (promise) (promise))` (call the underlying lambda).

Because of the reasons described in the section above, this would not work as intended, `delay` needs to be a special form.

One other idiosyncracy is that once a promise was forced, it remembers its value.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def a 1) (def promise (delay a)) (set! a 2) (force a) ; => 2 (set! a 3) (force a) ; => 2

5 Lists and Streams

In Lisp, **lists** are built by nesting `cons`.
`(list 1 2 3)` is just syntactic sugar for `(cons 1 (cons 2 (cons 3 '())))`.

Streams are lazy lists, their elements are computed on demand they can be infinitely long.

Let's start with a simple example:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def ones (cons 1 ones))

This doesn't work because at the time the body `(cons 1 ones)` is being evaluated, `ones` is not yet defined in the parent environment.

On the other hand,

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def ones (cons 1 (delay ones)))

works perfectly fine, because the evaluation of `ones` in the `cons` is being delayed until we actually access it (and evaluating `def` is completed).

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (fst ones) ; => 1 (rst ones) ; => promise(?), a promise that has not yet been evaluated (force (rst ones) ; => (1 . promise(?))

To keep the code as simple as possible, I added a special form `(stream-cons foo bar)` that is equivalent to `(cons foo (delay bar)` and a method `(stream-rst stream)`[1] that forces the `rst` of `stream`.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def ones (stream-cons 1 ones)) (stream-rst ones) ; => (1 . promise(?))

6 Working with Streams

Before we continue, it would be nice to have some way to display streams (up to some fixed, finite length).

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn stream-print (limit stream) (if (> limit 0) (do (println (fst stream)) (stream-print (dec limit) (stream-rst stream)))))

`stream-print` checks if the limit is greater than `0`, prints the `fst` of the stream and calls itself recursively with `(dec limit)` (`limit - 1`) and the forced `rst` (`stream-rst`) of the stream.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (stream-print 5 ones) ; 1 ; 1 ; 1 ; 1

To do some more interesting things, we need ways to combine and manipulate streams.

`stream-map` creates a new stream with the results of applying `fun` to the elements of `stream`.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn stream-map (fun stream) (stream-cons (fun (fst stream)) (stream-map fun (stream-rst stream))))

[1]There is no need to have something like `stream-fst`, the `fst` of the stream is not a promise.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (stream-print 4 (stream-map inc ones)) ; 2 ; 2 ; 2 ; 2

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn stream-combine (fun s1 s2) (stream-cons (fun s1 s2) (stream-combine fun (stream-rst s1) (stream-rst s2))))

(defn stream-add (s1 s2) (stream-combine + s1 s2))

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (stream-print 4 (stream-add ones ones)) ; 2 ; 2 ; 2 ; 2

Still pretty boring, how about a stream of natural numbers (excluding 0)?

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def natural-numbers (stream-cons 1 (stream-add natural-numbers ones))) (stream-print 4 natural-numbers) ; 1 ; 2 ; 3 ; 4 ; 5

This might be a little hard to understand, especially if you have never worked with streams or lazy evaluation before.

```
natural-numbers = 1, (natural-numbers[0] + 1), (natural-numbers[1] + 1),  ...
```

An alternative implementation that might be easier to understand:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn natural-numbers-from (n) (stream-cons n (natural-numbers-from (inc n))))

(def natural-numbers (natural-numbers-from 1))

7 Fibonacci Numbers

The Fibonacci Numbers are defined as

- `(fib 0) = 0`
- `(fib 1) = 1`
- `(fib n) = (+ (fib (- n 1)) (fib (- n 2)))`

This is equivalent to adding the `fib` stream its `rst`, a version of itself that is shifted by one.

```
  0 1 1 2 3 5  8 13 ...
+ _ 0 1 1 2 3  5  8 ...
= 0 1 2 3 5 8 13 21 ...
```

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def fibs (cons-stream 0 (cons-stream 1 (stream-add fibs (stream-rst fibs)))))

(stream-print 10 fibs) ; 0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13 ; 21 ; 34

8 Filtering Streams

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn stream-filter (pred stream) (if (pred (fst stream)) (stream-cons (fst stream) (stream-filter pred (stream-rst stream))) (stream-filter pred (stream-rst stream))))

Filtering over streams is more involed than `map`.
If the result of `(pred (fst stream))` is true, we construct a new stream with `(fst stream)` and the **delayed** result of filtering the rest of the stream. Otherwise, we call `stream-filter` again with the rest of the stream, skipping `fst`.

To better understand what is happening when forcing elements of the filtered stream, let's build a stream that displays a message when one of its elements is forced.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (def debug-stream (stream-map (fn (n) (do (print "Forcing ") (println n) n)) (natural-numbers-from 0))) ; Forcing 0

Only one line is being printed, the rest of the new stream will only be evaluated when we need it.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn is-multiple-of (div) (fn (n) (= 0 (

(def multiples-of-five (stream-filter (is-multiple-of 5) debug-stream))

(fst multiples-of-five) ; => 0

No new lines are being printed, the first element `0` is already a multiple of `5`.

When we try to output more elements of the stream, something strange happens:

```
(println (fst (stream-rst multiples-of-five)))
; Forcing 1
; Forcing 2
; Forcing 3
; Forcing 4
; Forcing 5
5
```

`stream-filter` starts forcing elements of the stream until it finds the next one that satisfies `pred`, in this case `5`.

9 An Infinite Prime Sieve

The Sieve of Erathosthenes works by taking a list of numbers (starting with 2), going to the first element, removing all its multiples from the list (4, 6, 8, . . .) and then repeating these steps with the next elements in the list (first 3, then 5) over and over again.

Once this process has reached the end of the list, only prime numbers remain.

Based on `(natural-numbers-from 2)` we can use the same algorithm to create a (infinite) stream of primes!

Start with removing all multiples of the first element from a stream:

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn is-not-multiple-of (div) (fn (n) (!= 0 (

(defn remove-multiples-of-first (stream) (stream-cons (fst stream) (stream-filter (is-not-multiple-of (fst stream)) (stream-rst stream))))

(stream-print 6 (remove-multiples-of-first (natural-numbers-from 2))) ; 2 ; 3 ; 5 ; 7 ; 9 ; 11 ; 13 ; 15 ; 17 ; 19

We are nearly there, the first few numbers are already primes but `9` is a multiple of `3`, `15` is a multiple of `5`, . . .

What is missing is the next step of the algorithm, > . . . and then repeating these steps with the next elements in the list > (first 3, then 5) over and over again.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn remove-multiples-of-first (stream) (stream-cons (fst stream) (remove-multiples-of-first (stream-filter (is-not-multiple-of (fst stream)) (stream-rst stream)))))

(def primes (remove-multiples-of-first (natural-numbers-from 2)))

(stream-print 10 primes) ; 2 ; 3 ; 5 ; 7 ; 11 ; 13 ; 17 ; 19 ; 23 ; 29

The change in the code was pretty minor but now `remove-multiples-of-first` applies itself recursively to the new, filtered stream, completing the prime sieve.

To make sure, we can define a method that returns the nth[2] element of a stream and check agains a list of primes.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn stream-nth (n stream) (if (= n 0) (fst stream) (stream-nth (dec n) (stream-rst stream)))) (stream-nth 49 primes) ; => 229, which is in fact the 50th prime

[2] `(stream-nth 49 primes)` is the 50th element of the stream because it is 0-indexed