Euler Lisp

Leon Rische

[2019-07-22 Mon 20:14]

Contents

- EulerLisp Benchmarks
- EulerLisp Language Documentation
- EulerLisp VM
- EulerLisp Compiler
- EulerLisp Object System [incomplete]

A LISP bytecode VM implemented in Rust.
Source code on Github: l3kn/EulerLisp
I always wanted to create my own programming language.

It might be one of the programming projects with the highest floor-to-ceiling ratio: a simple interpreter can take up less than a hundred lines of code, but adding features and improving the speed quickly increases the complexity, until it reaches the heights of industrial strength compilers like gcc, with millions of lines of code.

At around 7k lines of code this project lies somewhere inbetween (at least on a logarithmic scale).

The language is a Lisp1 modeled after Scheme and runs on a simple bytecode-VM, so that it is possible to write non-trivial programs and have them run at a reasonable speed.

language=scheme,label= ,caption= ,captionpos=b,numbers=none (defn fib (n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
(println (fib 30))

Of course this is a pretty meaningless benchmark, but on my machine the recursive fibonacci fuction above takes 1.1s to execute in EulerLisp which gets makes it between 2x and 10x slower than a few other languages I tested:

- Ruby, 0.11s
- Python, 0.23s
- Elixir, 0.48s
- Chicken Scheme (interpreted), 0.68s
- Chicken Scheme (compiled), 0.07s
- C (gcc -O3), 0.08s

1 Motivation

As an incentive to make the language fast and usable, I'm using it to work through some Project Euler math problems.

`https://projecteuler.net/profile/leonrische.png`

Up till now, I've solved around 200 of the problems and written around 7k lines of EulerLisp code.

TODO: List of favorite problems

2 Data Types

The main data type is called `Datum` and has different variants:

- `Bool`
- `Integer` (64bit signed)
- `Bignum` (multiple precision)

- `Rational` (64bit numerator and denominator)
- `Float` (64bit)
- `Char`
- `String`
- `Symbol` (pointer into a symbol table)
- `Vector`
- `Builtin` (reference to a function implemented in rust)
- `Closure` (reference to a function in the target language)
- `PriorityQueue` (for performance reasons, should be implemented in lisp)
- `Undefined`
- `Nil`

3 Builtin Functions

3.1 Bitwise

- `(bitwise-and args*)`
- `(bitwise-or args*)`
- `(bitwise-xor args*)`
- `(bitwise-not args)`

3.2 Comparison

- `(` args*)=, numeric equality
- `(!` a b)=, numeric inequality
- `(equal? args*)`, object equality
- `(< args*)`
- `(<` args*)=

- `(> args*)`
- (> args*)=
- `(min args*)`
- `(max args*)`

3.3 Lists, Vectors, Pair

- `(cons a b)`, create a pair from `a` and `b`
- `(fst p)`, get the first element of a pair
- `(rst p)`, get the second element of a pair
- `(set-fst! p v)`, set the first element of a pair
- `(set-rst! p v)`, set the second element of a pair
- `(list args*)`, create a list, equivalent to `(cons arg1 (cons arg2 (cons ... (cons argn '()))))`
- `(list->vector lst)`
- `(permutations lst)`
- `(combinations lst len)`
- `(sort lst)`
- `(uniq lst)`
- `(join str lst)`
- `(vector args*)` create a vector
- `(vector-ref vec i)`
- `(vector-set! vec i v)`
- `(vector-push! vec v)`
- `(vector-pop! vec)`
- `(vector-shuffle! vec)`

- `(vector-delete! vec i)`
- `(vector-length vec)`
- `(vector->list vec)`
- `(make-vector size [initial])`
- `(vector-copy vec [from] [to])`

3.4 Numbers, Math

- `(+ args*)`
- `(- arg)`, negation
- `(- args*)`, subtraction
- `(* args*)`
- `(/ args*)`
- `(% a mod)`
- `(div a mod)`, integer division
- `(divmod a mod)`, the result is a pair `(quotient . remainder)`
- `(powf n e)`,
$$n^e$$
(`pow` for non-integer exponents)
- `(sqrt n)`,
$$n^{\frac{1}{2}}$$
- `(cbrt n)`,
$$n^{\frac{1}{3}}$$
- `(ln a)`, logarithm base
$$e$$
- `(log2 a)`, logarithm base
$$2$$

- `(log10 a)`, logarithm base 10
- `(log a base)`
- `(ceil a)`
- `(round a)`
- `(floor a)`
- `(» a by)`, right shift
- `(« a by)`, left shift
- `(popcount a)`, count the number of 1s in the binary representation of a
- `(prime? a)`
- `(zero? a)`
- `(number->digits a)`, list of the digits of `a` in reverse order
- `(digits->number lst)`
- `(number-of-digits a)`
- `(denominator a)`
- `(numerator a)`
- `(sin a)`, `(cos a)`, `(tan a)`
- `(asin a)`, `(atan a)`, `(acos a)`
- `(atan2 a b)`, four quadrant inverse tangent
- `(radiants a)`

- `(totient a)`, `(totient-sum a)`,

  $$\varphi(a)$$

  and

  $$\sum_{i=1}^{a} \varphi(i)$$

- `(modexp b e n)`,

  $$b^e \mod n$$

- `(modular-inverse b n)`, inverse of

  $$b$$

  modulo

  $$n$$

- `(extended-euclidian a b)`, a list

  $$(cde)$$

  , so that

  $$ca + db = d = gcd(a, b)$$

- `(prime-factors a)`, prime factors of `a` as a list of pairs `(p . e)`
- `(num-prime-factors a)`, number of prime factors of `a`
- `(primes n)`, a list of the first `n` primes
- `(rand from to)`, random number in

  $$[from, to]$$

3.5 Bignum

- `(bignum n)`, convert `n` to a bignum
- `(digits->bignum digits)`, create a bignum from a list of digits (in reverse order)
- `(bignum-chunks bn)`, base 10^9 "digits" of the bignum
- `(chunks->bignum bn)`, create a bignum from a list of chunks

3.6 Input / Output

TODO

3.7 String

TODO

3.8 Types

TODO

4 Resources

If this inspired you to start building your own programming language, here are some resources that helped me:

4.1 Background

- The Roots of Lisp

4.2 Algorithms & Data Structures

- The Art of Computer Programming
- Introduction to Algorithms

4.3 Implementation

- Lisp In Small Pieces

4.4 Specs

- R5RS
- R6RS
- R7RS
- The Scheme Programming Language

5 TODO

5.1 Garbage Collection

Currently I'm using rusts reference counted pointers `Rc` and `RefCell` as a substitute for implementing my own garbage collection.

This works fine in most cases, but because Scheme allows creating circular lists it it possible to create objects that are not reachable by some root but have non-zero reference counts.

language=scheme,label= ,caption= ,captionpos=b,numbers=none (defn fill-memory () (let ((circular (cons 1 2))) (set-rst! circular circular)) (fill-memory))

(fill-memory)

In the future it would be nice to write my own simple garbage collector.

6 Continuations

Continuations capture the remaining parts of a computation.

The VM maintains the following state components:

- val register
- fun register
- arg1 register
- arg2 register
- env
- env stack
- stack
- program counter (through a `Bytecode` struct)
- program counter stack (through a `Bytecode` struct)

`call-with-current-continuation` (often abbreviated as `call/cc`) captures the current evaluation context and turns it into an object that can be called.

`(call/cc f)` allocates an activation frame, fills it with an object representing the current continuation and then calls the `f` with this activation frame.

We can ignore the registers, they are not saved in any function call. The program counter can be ignored, too, as it is restored by the `RETURN` in the body of function `f`.

Continuation invocation works by restoring the stack, setting `val` to the value the continuation was called with and then continuing to run the program.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defn f (cont) (cont 2) 3)

(println (f id)) (println (call/cc f))

```
   CREATE-CLOSURE 1
   JUMP @510
   PUSH-SHALLOW-ARGUMENT-REF 0
   PUSH-CONSTANT $2
   FUNCTION-INVOKE tail: false, arity: 1
   RESTORE-ENV
   CONSTANT $3
   RETURN
510:
   GLOBAL-SET g322
   // (set! f ...)

   PUSH-CHECKED-GLOBAL-REF println
   PUSH-CHECKED-GLOBAL-REF g322
   PUSH-CHECKED-GLOBAL-REF g236
   FUNCTION-INVOKE tail: false, arity: 1
   // call (f id)
   RESTORE-ENV
   PUSH-VALUE
   FUNCTION-INVOKE tail: false, arity: 1
   // call (println res)
   RESTORE-ENV
   PUSH-CHECKED-GLOBAL-REF println
   CHECKED-GLOBAL-REF g322
   // load ‘f‘ closure into val
   CALL-CC
   // call ‘f‘ with current continuation as argument
   PUSH-VALUE
   FUNCTION-INVOKE tail: true, arity: 1
```

When `FUNCTION-INVOKE` in the closure is called on the continuation,

TODO: Call/cc as builtin function, so that I can use it in map