Meta-Circularity, and Vice-Versa

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/˜didier

ACCU 2011 – Thursday, April 14th

Table of contents

Meta-circularity

- **Self-interpreter** (meta-interpreter):
  written in the language it interprets
  - Bootstrapping problem
  - But not so surprising for a turing-complete language
- **Meta-circular evaluator**: special case where the language is restated in terms of itself (no additional implementation required)
- **Homoiconicity** (code is data):
  Program representation in a primitive data structure

The essence of an interpreter

- **Expression evaluation**: evaluate the arguments, then *apply* the operator's value to their evaluation
- **Operator application**: augment the environment with the formal parameters, then *evaluate* the operator's value

Remarks

- Strict evaluation ($\neq$ lazy)
- Applicative order
- Substitution model
- Cf. λ-calculus (α-conversion and β-reduction)

Lisp Nativity

- **MIT, Artificial Intelligence Laboratory, 1958**
  - Project "Advice Taker"
  - John McCarthy's founding paper: "Recursive Functions of Symbolic Expressions and their Computation by Machine"
  - IBM 704
- **Intentional simplifications**
  - Apply and eval mixed up
  - Lisp dialect modernized
  - No error checking (and stuff)

IBM 704

IBM 704 ELECTRONIC DATA-PROCESSING MACHINES

S-expressions (Sexp)

- Sexp = Atom or list of expressions
- Atom = sequence of letters
- List = parenthesized, space-separated sequence of expressions

Examples

```
foo
()
(foo)
(foo bar)
((the cat) (eats (the mouse)))
```

Sexp values

- Mathematical background (Cf. λ-calculus)
- *Pure* functional programming
- Atoms: self-evaluating or environmental
- Non atomic sexp: `(OP ARG1 ARG2 ...)` where
  - `OP` is an *operator*
  - `ARGx` is an *argument*
- 7 primitive operators (axioms)

Operator #1: quote

- Returns its argument unevaluated
- Syntactic sugar: `'x`

Examples

```
> (quote a)
a

> '(a b c)
(a b c)

> (quote '(a b c))
(quote (a b c))
```

Operator #2: `atom`

- Predicate
- Returns `t` (true) if its argument is an atom
- Returns `nil` (false; *nihil*; `()`) otherwise

Examples

```
> (atom 'a)
t

> (atom '(a b c))
nil

> (atom ())
t
```

Digression: argument evaluation

- `atom` evaluates its argument ($\neq$ `quote`)

Examples

```
> (atom (atom 'a))
t

> (atom '(atom 'a))
nil
```

- `quote` is LISP-specific.
- code $\Longleftrightarrow$ data
- Structural reflexivity
- **But note**: DANGER, WILL ROBERTSON !!

Operator #3: eq

- Equality operator
- Returns `t` if both arguments are the same atom
- Returns `nil` otherwise

Examples

```
> (eq 'a 'a)
t

> (eq 'a 'b)
nil

> (eq nil ())
t

> (eq '(a b c) '(a b c))
nil
```

Operators #4 and #5: `car` and `cdr`

- `car` returns the first element of a list
- `cdr` returns the rest of a list

Examples

```
> (car '(a b c))
a

> (cdr '(a b c))
(b c)

> (car ())
nil

> (cdr nil)
nil
```

The truth about `car`, `cdr` and lists

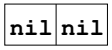
- IBM 704's hardware architecture
- `car`: **C**ontents of **A**ddress **R**egister
- `cdr`: **C**ontents of **D**ecrement **R**egister
- A list only has a `car` and a `cdr`
- The `car` and the `cdr` are separated by a dot `(car . cdr)`
- The space notation is just syntactic sugar

Some examples

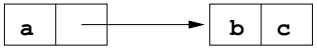
~~()~~

(a b) <=> (a . (b)) <=> (a . (b . nil))

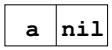
(nil . nil)



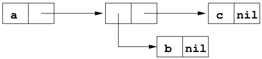
(a b . c) <=> (a . (b . c))



(a) <=> (a . nil)



(a (b) c) <=> (a . ((b) c)) <=> (a . ((b) . (c)))
<=> (a . ((b . nil) . (c . nil)))



Operator #6: `cons`

- Canonical list operator
- Constructs a list by car and cdr

Examples

```
> (cons 'a '(b c))
(a b c)

> (cons 'a (cons 'b (cons 'c ())))
(a b c)

> (car (cons 'a '(b c)))
a

> (cdr (cons 'a '(b c)))
(b c)
```

Operator #7: `cond`

- Conditional branching
- Multiple test/body clauses

Examples

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a)  'second))
second

> (cond ((eq 'a 'b) 'first)
        ((eq 'c 'd) 'second)
        (t          'default))
default
```

Summary

- 7 primitive operators
- 5 of them always evaluate their arguments
- The 2 others are `quote` and `cond`
- Distinction function / special operator

Functional denotation

The mathematical term “function” is ambiguous:

- Both *forms* $x + y^2$ and $f(x, y)$ are called “functions”
- But $x + y^2$(3, 4) = 13 or 19?
- Church’s λ notation (1941):
  $\lambda((x_1, \ldots, x_n), \varepsilon)$
  $\lambda((x, y), x + y^2)(3, 4) = 19$

Application to LISP

- **LISP functions:**
  - `(lambda (`p_1 `...`p_n`)` e`)`
  - p_i are atoms (parameters)
  - e is an sexp
- **LISP function call:**
  - `((lambda (`p_1 `...`p_n`)` e`)` a_1 `...`a_n`)`
  - a_i are evaluated
  - e is evaluated with every p_i substituted with a_i's value
- **Note**: The value of a_i may very well be a function…

Examples


```
> ((lambda (x) (cons x '(b c)))
    'a)
(a b c)

> ((lambda (x y) (cons x (cdr y)))
    'z '(b c))
(z c)

>  ((lambda (f) (f '(b c)))
     '(lambda (x) (cons 'a x)))
>> ((lambda (x) (cons 'a x))
     '(b c))
(a b c)
```

Recursive functions

- **The lambda notation is inadequate** (Rochester):

```
fact = (lambda (n) (* n (fact??? (-1 n))))
```

- **New denotation:**
  - (label f (lambda (p_1 ...p_n) e))
  - (defun f (p_1 ...p_n) e)
  - Same behavior as a lambda-expression, but every occurrence of `f` evaluates to the lambda-expression itself.

Example

```
(defun fact (n) (* n (fact (-1 n))))
```

Convenience shortcuts

- **(cadr** e): `(car (cdr e))`
- **(cdar** e): `(cdr (car e))`
- **(c[ad]+r** e): ...
- **(list** $e_1 \dots e_n$): `(cons` e_1 `...(cons` e_n `()))`

Examples

```
> (cadr '((a b) (c d) e))
(c d)

> (cdar '((a b) (c d) e))
(b)

> (list 'a 'b 'c)
(a b c)
```

Logical operators

- **null**:

```
(defun null (x)
  (eq x nil))
```

- **not**:

```
(defun not (x)
  (cond (x nil)
        (t t)))
```

- **and**:

```
(defun and (x y)
  (cond (x (cond (y t)
                 (t nil)))
        (t nil)))
```

append

```
(defun append (x y)
  (cond ((null x) y)
        (t (cons (car x) (append (cdr x) y)))))
```

Examples

```
> (append '(a b) '(c d))
(a b c d)

> (append nil '(a b))
(a b)

> (append '(nil) '(a b))
(nil a b)
```

pair

Create an “association list” (alist) from two lists

```
(defun pair (x y)
  (cond ((and (null x) (null y))
         nil)
        ((and (not (atom x)) (not (atom y)))
         (cons (list (car x) (car y))
               (pair (cdr x) (cdr y))))))
```

Examples

```
> (pair '(a b c) '(d e f))
((a d) (b e) (c f))
```

assoc

Return the `cadr` of the first matching alist entry

```
(defun assoc (x y)
  (cond ((eq x (caar y)) (cadar y))
        (t (assoc x (cdr y)))))
```

Examples

```
> (assoc 'x '((x a) (y b)))
a

> (assoc 'x '((x one) (x two) (y b) (x three)))
one
```

LISP can be written in itself

Stephen B. Russell and Daniel J. Edwards

Here it is...

```
(defun eval (exp env)
  (cond
   ((atom exp) (assoc exp env))
   ((atom (car exp))
    (cond ((eq 'quote (car exp)) (cadr exp))
          ((eq 'atom  (car exp)) (atom (eval (cadr exp) env)))
          ((eq 'eq    (car exp))
           (eq (eval (cadr exp) env) (eval (caddr exp) env)))
          ((eq 'car   (car exp)) (car (eval (cadr exp) env)))
          ((eq 'cdr   (car exp)) (cdr (eval (cadr exp) env)))
          ((eq 'cons  (car exp))
           (cons (eval (cadr exp) env) (eval (caddr exp) env)))
          ((eq 'cond  (car exp)) (condeval (cdr exp) env))
          (t (eval (cons (assoc (car exp) env) (cdr exp)) env))))
   ((eq (caar exp) 'label)
    (eval (cons (caddar exp) (cdr exp))
          (cons (list (cadar exp) (car exp)) env)))
   ((eq (caar exp) 'lambda)
    (eval (caddar exp)
          (append (pair (cadar exp) (listeval (cdr exp) env)) env)))))
```

Explanations

- `eval` takes two arguments:
  - `exp` is an sexp to evaluate
  - `env` is the evaluation environment
    (alist of atoms and their corresponding values)
- `eval` has 4 evaluation clauses:
  - atoms
  - lists (beginning with an atom)
  - `label` expressions
  - `lambda` expressions

Clause #1: atom

Seek out its value in the environment

```
((atom exp) (assoc exp env))
```

Example

```
> (eval 'x '((a b) (x val)))
>> (assoc 'x '((a b) (x val)))
val
```

Clause #2: `(op ...)` (1/2)

`op` is a primitive operator: call the operator on the evaluation of its arguments

```
((eq 'cons (car exp))
 (cons (eval (cadr  exp) env)
       (eval (caddr exp) env)))
```

Example

```
> (eval '(cons x '(b c)) '((x a)))
>> (cons (eval x '((x a)))
         (eval (quote (b c)) '((x a))))
(a b c)
```

Exceptions

- `quote` does not evaluate its argument:

```
((eq 'quote (car exp))
 (cadr exp))
```

- `cond` is also treated in a special way (lazy):

```
((eq 'cond (car exp))
 (condeval (cdr exp) env))
```

condeval

```
(defun condeval (conds env)
  (cond ((eval (caar conds) env)
         (eval (cadar conds) env))
        (t
         (condeval (cdr conds) env))))
```

Clause #2: (op ...) (2/2)

op is an atom: replace it by its value (must be a label or lambda) and evaluate the call

```
(t (eval (cons (assoc (car exp) env)
               (cdr exp))
         env))
```

Examples

```
> (eval '(f '(b c)) '((f (lambda (x) (cons 'a x)))))
>> (eval ((lambda (x) (cons 'a x)) '(b c)) '((f (lambda (x) (cons 'a x)))))
(a b c)
```

Clause #3: labels

((label f (lambda ($p_1 \ldots p_n$) e)) $a_1 \ldots a_n$)
Add the lambda-expression in the environment, and evaluate it

```
((eq (caar exp) 'label)
 (eval (cons (caddar exp) (cdr exp))
       (cons (list (cadar exp) (car exp)) env)))
```

> (eval ((label f (lambda ($p_1 \ldots p_n$) e)) $a_1 \ldots a_n$)
      env)
>> (eval ((lambda ($p_1 \ldots p_n$) e) $a_1 \ldots a_n$)
      ((f (label f (lambda ($p_1 \ldots p_n$) e))) env))

Clause #4: lambdas

`((lambda` $(p_1 \ldots p_n)$ e`)` $a_1 \ldots a_n$`)`

Evaluate e with associations `(`p_i `(eval` a_i`))` added to the environment

```
((eq (caar exp) 'lambda)
 (eval (caddar exp)
       (append (pair (cadar exp)
                     (listeval (cdr exp) env)) env)))
```

`> (eval ((lambda` $(p_1 \ldots p_n)$ e`)` $a_1 \ldots a_n$`) env)`

`>> (eval` e `((`p_1 $\overline{a_1}$`)` ... `(`p_n $\overline{a_n}$`) env))`

listeval

Returns the list of all arguments evaluated in the environment

$$(a_1 \ldots a_n) \Longrightarrow (\overline{a_1} \ldots \overline{a_n})$$

```
(defun listeval (args env)
  (cond ((null args) nil)
        (t (cons (eval (car args) env)
                 (listeval (cdr args) env)))))
```

It's even simpler than that (but less readable)

- `eval` only requires the 7 primitive operators
- But don't forget to add `exp` and `env` to the environment!

Example

```
(assoc (car exp) env)

(eval '((label assoc (lambda (x y)
                      (cond ((eq (car (car y)) x)
                             (car (cdr (car y))))
                            (t (assoc x (cdr y)))))))
        (car exp) env)
      (cons (cons 'exp (cons exp ()))
            (cons (cons 'env (cons env ())) env)))
```

The `label` notation is unnecessary

- Anonymous recursion possible
- Idea: pass the (anonymous) recursive function as a parameter!

Let's switch to Scheme syntax…

Factorial function

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

Passing the procedure as a parameter…

fact-maker, take 1

```
(define fact-maker
  (lambda (procedure)
    (lambda (n)
      (if (zero? n)
          1
          (* n (procedure (- n 1)))))))
```

- Hoping for: `((fact-maker fact-maker) 5)`…
- to work. **NOT!**

Let's try again…

fact-maker, take 2

```
(define fact-maker
  (lambda (procedure)
    (lambda (n)
      (if (zero? n)
          1
          (* n ((procedure procedure) (- n 1)))))))
```

- `((fact-maker fact-maker) 5) => 120`
- It works! But we still have a name…

Using the lambda on itself directly. . .

That's it at last

```
(((lambda (procedure)
    (lambda (n)
      (if (zero? n)
          1
          (* n ((procedure procedure) (- n 1))))))
  (lambda (procedure)
    (lambda (n)
      (if (zero? n)
          1
          (* n ((procedure procedure) (- n 1)))))))
 5)
```

- But can we generalize to *any* function?

Abstracting `procedure` away (1/3)

(procedure procedure) is boring

```
(lambda (n)
  (if (zero? n)
    1
    (* n ((procedure procedure) (- n 1)))))
```

A nice little trick

```
f         <=>  (lambda (x) (f x))
(f arg)  <=> ((lambda (x) (f x)) arg)
```

Abstracting `procedure` away (2/3)

Let's plug the trick in

```
(lambda (n)
  (if (zero? n)
    1
    (* n ((lambda (arg) ((procedure procedure) arg)) (- n 1))))))
```

And parametrize the function

```
((lambda (func)
   (lambda (n)
     (if (zero? n)
       1
       (* n (func (- n 1))))))
  (lambda (arg) ((procedure procedure) arg))))
```

Abstracting procedure away (3/3)

Let's plug the new form in

```
(((lambda (procedure)
    ((lambda (func)
       (lambda (n)
         (if (zero? n)
           1
           (* n (func (- n 1))))))
      (lambda (arg) ((procedure procedure) arg))))
  (lambda (procedure)
    ((lambda (func)
       (lambda (n)
         (if (zero? n)
           1
           (* n (func (- n 1))))))
      (lambda (arg) ((procedure procedure) arg)))))
 5)
```

Where is `fact` now ?

Fact is here:

```
(define dofact
  (lambda (func)
    (lambda (n)
      (if (zero? n)
        1
        (* n (func (- n 1)))))))
```

And we can just plug it in:

```
(define fact
  ((lambda (procedure)
     (dofact (lambda (arg) ((procedure procedure) arg))))
   (lambda (procedure)
     (dofact (lambda (arg) ((procedure procedure) arg))))))
```

Bingo!

The famous Y combinator:

```
(define Y
  (lambda (X)
    ((lambda (procedure)
       (X (lambda (arg) ((procedure procedure) arg))))
     (lambda (procedure)
       (X (lambda (arg) ((procedure procedure) arg)))))))
```

And the final factorial:

```
(define fact (Y dofact))
```

Reminder: dofact was not so difficult to get

```
(define dofact
  (lambda (func)
    (lambda (n)
      (if (zero? n)
        1
        (* n (func (- n 1)))))))
```

Dynamic Scoping

What's the real name of this function?

```
(defun mysterious (func lst)
  (let (elt n)
    (while (setq elt (pop lst))
      (push (funcall func elt) n))
  (nreverse n)))

(defun increment-list (n lst)
  (mysterious (lambda (elt) (+ elt n))
              lst))
```

- Very first example of higher-order function
- In McCarthy's original paper (1958)
- Doesn't work ;-)

What Lisp brought to the picture

- **From the λ-calculus**
  - Homoiconicity
  - Meta-circularity
  - Structural reflexivity
  - Functional paradigm
  - Expressions
  - Recursion
  - Dynamic typing
- **And also** (less relevant)
  - Conditional branches
  - Garbage collection

Practical uses (Common Lisp)

- `eval`, `read`, `print`
- `funcall`, `apply`
- `compile`
- debugger
- macros, reader macros, compiler macros
- meta-object protocols
- *etc.*