About Lisp

...or Lambda, the ultimate lecture

Yoni Rabkin

yonirabkin@member.fsf.org

Abstract

First we shall introduce symbolic, conditional and meta expressions and their recursive definitions with the λ-notation. Then we will briefly describe how such expressions might be represented by a computer. We shall introduce the Lisp REPL and use it to explore a number of basic Lisp paradigms such as closures, functional programming and Lisp macros. Finally we shall look at the past and present of Lisp as a language.

What is Lisp?

What is Lisp? What can we say generally about Lisp?

- *"Lisp is a programmable programming language"*[(a)]
- There are many Lisps, standardised and not.
- Lisp has very little syntax.
- Lisp's roots are in the mathematical representation of recursive functions [2].
- Doesn't have to look like: $\lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx))$

[(a)] John Foderaro, CACM, September 1991

Conditional Expressions

$$(p_1 \to e_1, \cdots, p_n \to e_n)$$

It may be read, "if p_1 then e_1 otherwise if p_2 then e_2,$\cdots$, otherwise if p_n then e_n" where the p's are propositional expressions and the e's are expressions of any kind.

$$(1 < 2 \to 4, 1 > 2 \to 3) = 4$$

$$(2 < 1 \to 4, T \to 3) = 3$$

$(2 < 1 \to 3, 4 < 1 \to 4)$ is undefined

$$|x| = (x < 0 \to -x, T \to x)$$

Recursive Function Definitions

$$n! = (n = 0 \to 1, T \to n \cdot (n-1)!)$$

$$sqrt\,(a, x, \epsilon) = \left(|x^2 - a| < \epsilon \to x, T \to sqrt\left(a, \frac{1}{2}\left(x + \frac{a}{x}\right), \epsilon\right)\right)$$

Church's λ-notation

Church's λ-notation [2] helps distinguish between a function and a form. The expression $y^2 + x$ is a form. A form is converted to a function once the correspondence between variables occuring the form and an ordered list of arguments is determined.

Church's λ-notation II

If ε is a form in variables $x_1, \cdots, x_n$ then $\lambda\left(\left(x_1, \cdots, x_n\right), \varepsilon\right)$ will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables $x_1, \cdots, x_n$ in that order in ε and evaluating the resulting expression.

$$\lambda\left((x, y), y^2 + x\right)(3, 4) = 19$$

The concept of a *label* is used to define recursive functions.

$$label\left(a, \varepsilon\right)$$

$$label\left(fact, \lambda\left((n), (n = 0 \rightarrow 1, T \rightarrow n \cdot \left(fact\left(n - 1\right)\right)\right)\right)\right)$$

Symbolic Expressions

- S-expressions are formed out of:
  - $($
  - $)$
  - .
  - infinite set of atomic symbols

1. Atomic symbols are S-expressions
2. if e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$

Symbolic Expressions II

A list of arbitrary length written as

$$(m_1, m_2, \cdots, m_n)$$

is represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))$$

Meta Expressions

Since we cannot describe S-expressions using S-expressions, we use M-expressions to describe primitive functions.

$$cons[A; B] = (A \cdot B)$$

$$cons[cons[A; B]; C] = ((A \cdot B) \cdot C)$$

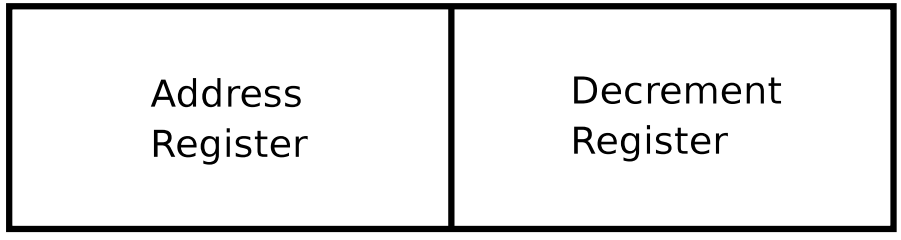
if x and y represent any two S-expressions, the following identities are true:

$$car[cons[x; y]] = x$$

$$cdr[cons[x; y]] = y$$

Representation of List Structure: Cells

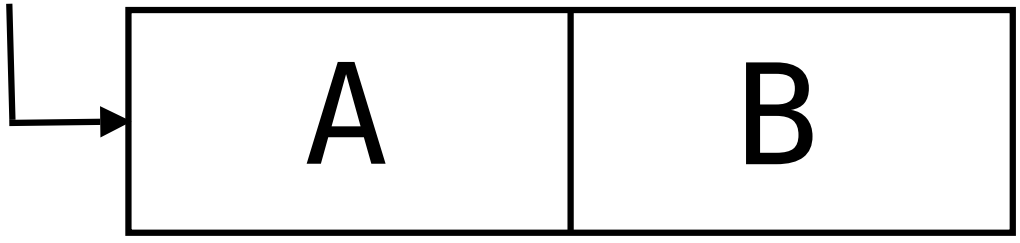
Lisp lists are typically represented using pointers to pairs of units (aka cells) of memory. The exact representation varies throughout architectures.



The first half of the cell is called the *contents of the address register* or *CAR*. The second half of the cell is called the *contents of the decrement register* or *CDR*.[4]

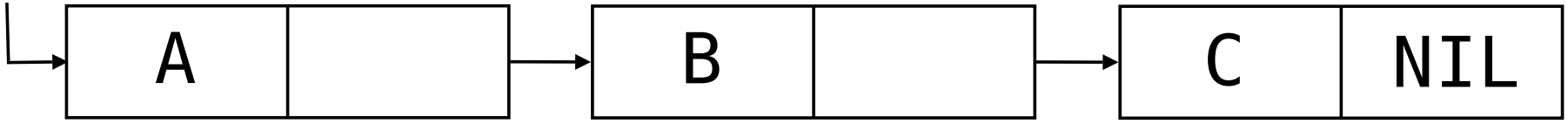
Representation of List Structure: Cons

The S-expression $(A \cdot B)$ is represented as a cons-cell whose *CAR* is the atomic symbol A and whose *CDR* is the atomic symbol B.



Representation of List Structure: Lists

The S-expression $(A\ B\ C)$ is represented as a chain of cells. The *CAR* of the first cell contains the atomic symbol A, the *CDR* contains a pointer to the *CAR* of the next cell, which contains the atomic symbol B and so on... Finally, the list is terminated by the atomic symbol NIL in the *CDR* if the last cell.



To the Metal

$$\lambda\left(\left(x,y\right),y^2+x\right)$$

```
CL-USER> (lambda (x y) (+ (* y y) x))
  => #<FUNCTION (LAMBDA (X Y)) {B1B2ADD}>
CL-USER> (disassemble (lambda (x y)
                        (+ (* y y) x)))
; disassembly for (LAMBDA (X Y))
; MOV EDX, [EBP-4] ; entry point
; MOV EDI, [EBP-4]
; CALL #x100023F   ; GENERIC-*
; JNB L0
; MOV ESP, EBX
; MOV EDI, [EBP-8]
; CALL #x1000158   ; GENERIC-+
...
```


A Model for Evaluation

$$\lambda\left((x,y),y^2+x\right)(1+2,4)=19$$

```
CL-USER> ((lambda (x y)
           (+ (* y y) x))
         (+ 1 2) 4)
  => 19
```

$$\lambda\left((a,b),\lambda\left((x,y),y^2+x\right)(a,b)\right)(3,4)=19$$

```
CL-USER> (let ((x 3) (y 4))
           ((lambda (a b)
              (+ (* b b) a))
            x y))
  => 19
```


A Word about Prefix Notation

Lisp's prefix notation and s-expressions have some trivial advantages.

```
CL-USER> (let ((g 7) (c 3))
           (< 1 2 c 4 5 6 g 8))
=> T
```

...and some non-trivial implications such as representing parse-tree form.

`1 + 2 + 3`

is compiled to

`(+ 1 (+ 2 3))`

which in Lisp is shorthand for

`(+ 1 2 3)`

when you write Lisp, you are writing closer to the compiler

The REPL and Quotation

The *read-eval-print loop* or REPL, is the heart of an interactive Lisp environment. The REPL evaluates S-expressions and returns the result, which is also an S-expression.

```
CL-USER> (list 1 2 3 4)
=> (1 2 3 4)
CL-USER> '(list 1 2 3 4)
=> (LIST 1 2 3 4)
CL-USER> (cons '+ (list 1 2))
=> (+ 1 2)
CL-USER> (+ 1 2)
=> 3
CL-USER> 3
=> 3
```


A Rose by any other Name

$$label\left(fact, \lambda\left((n), (n = 0 \rightarrow 1, T \rightarrow n \cdot (fact\,(n-1)))\right)\right)$$

```
CL-USER> (defun fact (n)
           ((lambda (x)
              (cond ((= x 0) 1)
                    (t (* x (fact (- x 1))))))
            n))

CL-USER> (defun fact (x)
           (cond ((= x 0) 1)
                 (t (* x (fact (- x 1))))))

CL-USER> (fact 5)
  => 120
```


Closures

A combination of a function and a set of variable bindings[(a)] is called a closure[5].

```
CL-USER> (let ((n 0))
           (defun serial ()
             (cons (random 1000)
                   (incf n))))

CL-USER> (serial)
  => (715 . 1)

CL-USER> (serial)
  => (582 . 2)

...
```

(a) For why this is problematic, see [9]

Functions as arguments

The function:

```
(defun sum-cubes (a b)
  (if (> a b)
      0
      (+ (cube a)
         (sum-cubes (+ a 1) b))))
```

...is just a special case of:

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

...where:

$$f(n) = n^3$$

[1]

Functions as arguments II

```
(defun sum (term a next b)
  (if (> a b)
      0
    (+ (funcall term a)
       (sum term (funcall next a) next b))))

(defun cube (n)
  (* n n n))

(defun sum-cubes (a b)
  (sum #'cube a #'1+ b))

CL-USER> (sum-cubes 1 10)
  => 3025
```

[1]

Functions as return values

If $Dg(x)$ is the derivative of g evaluated at x:

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

...then we can express the idea as:

```
CL-USER> (let ((dx 0.001))
           (defun deriv (g)
             (lambda (x)
               (/ (- (funcall g (+ x dx))
                     (funcall g x))
                  dx))))
```

[1]

Functions as return values II

So given:

```
(defun square (n) (* n n))

(defun cube (n) (* n n n))

(defun identity (n) n)
```

...we get:

```
CL-USER> (funcall (deriv #'cube) 5)
  => 75.01221

CL-USER> (funcall (deriv #'square) 5)
  => 10.000229

CL-USER> (funcall (deriv #'identity) 5)
  => 0.99992746
```


Lisp Macros

Lisp macros transform one form into another while controlling the evaluation process using the full expressive power of the language.

- write domain-specific languages
- move computation to compile-time
- write special-forms which break the normal evaluation regime
- create opaque, ungrokable code and incite nervous break-down

Lisp Macros - ruining things

Let us implement two misfeatures at once: *infix/prefix inconsistency* and *operator overloading*
Our macros will have three parts:[(a)]

- a “defmacro” declaration
- computation done by the macro at compile time
- form to be evaluated at runtime

[(a)] this is true in the same sense that atoms consist of tiny blue and red spinning balls; it is a useful pedagogical lie

Lisp Macros - runtime

```
(defmacro misfeatures-I (form)
  (destructuring-bind (a op b) form
    '(funcall
      (cond ((and (numberp ,a)
                  (numberp ,b)) #'+)
            ((and (stringp ,a)
                  (stringp ,b)) #'concat)
            (t (error "unknown")))
      ,a ,b)))
```


Lisp Macros - runtime II

```
CL-USER> (misfeatures-I (1 + 2))
  => 3

CL-USER> (misfeatures-I ("1" + "2"))
  => "12"

CL-USER> (let ((foo 1) (bar 2))
           (misfeatures-I (foo + bar)))
  => 3

CL-USER> (let ((foo "1") (bar "2"))
           (misfeatures-I (foo + bar)))
  => "12"
```


Lisp Macros - compile-time

```
(defmacro misfeatures-II (form)
  (destructuring-bind (a op b) form
    (let ((type
           (cond ((and (numberp a)
                       (numberp b)) #'+)
                 ((and (stringp a)
                       (stringp b)) #'concat)
                 (t (error "unknown")))))
      `(funcall ,type ,a ,b))))
```


Lisp Macros - compile-time II

```
CL-USER> (misfeatures-II (1 + 2))
  => 3

CL-USER> (misfeatures-II ("1" + "2"))
  => "12"

CL-USER> (let ((foo 1) (bar 2))
           (misfeatures-II (foo + bar)))
  => ERROR!
```


Lisp History

A short and non-conclusive chronology of Lisp [6]. If this part seems a bit lengthy, it is because Lisp is very old!

- *1956* — Early thoughts (at Dartmouth)[(a)]
- *1958* — First implementation
- *1960-1965* — Lisp 1.5 (on an IBM 7094) and Lisps on the PDP-1 ,PDP-6, PDP-10

All Lisps were still basically identical until 1965. The PDP-6 and PDP-10 were well suited for Lisp with 36-bit words, 18-bit addresses and half-word instructions. [7]

[(a)] http://www-formal.stanford.edu/jmc/history/lisp/node2.html

Lisp History II

- *1960's until the early 1980's* — MacLisp[a] on the PDP-10 and Honeywell 6180 (under Multics)

MacLisp would later become Zetalisp on the Lisp-Machine and Scheme. MacLisp had numeric functions on par with the FORTRAN of the time. [b] This was a "high-water mark" to be revisited only recently.
The MacLisp community was very "open source" in spirit.

- *Same era as MacLisp* — Interlisp on PDP-10's, Vaxen and specialised XEROX Lisp machines.

---

[a] Mac == MIT's Project MAC, started 1963

[b] Fateman, Richard J. in a reply to editorial ACM SIGSAM Bulletin 25, pp. 9-11 March 1973

Lisp History III

- *1975* — Scheme, the Actor model of computation (now called “continuations” in Scheme), lexical scoping and closures
- *mid 1970’s* — Lisp360 and Lisp370 (later called Lisp/VM) on the IBM 360 and IBM 370
- *1974-1978* — MIT Lisp machines: CONS, CADR, LMI inc. and the Symbolics 3600
- *1973-1980* — Xerox Lisp machines, the Alto[(a)] and the “D-machines”: Dorado, Dolphin and Dandelion

---

[(a)] The Alto was used to build the first Smalltalk environment.

Lisp History IV

- *Early 1980's* — Franz Lisp[(a)] for UNIX
- *1981* — Emacs Lisp inside GNU/Emacs under UNIX
- *1982* — T under UNIX and VMS
- *1984* — CLtL I: Common Lisp by Steele, Fahlman, Moon, Weinreb and Gabriel
- *1986* — X3J13: Common Lisp ANSI standard

[(a)] Franz Inc. still exist today

Lisp Present: Scheme

- Created by Gerald Jay Sussman and Guy Steele
- Inspired by the *Actors* model of computation
- Minimalistic[(a)]
- Standard, but still being standardised (IEEE, R6RS, ANSI etc.)
- Inexplicably... GNU/Guile

[(a)] The complete Scheme standard is smaller than the index of CLtL2

Lisp Present: Common Lisp

- Created by the Common Lisp standards group
- The canonical language for modern Lisp applications
- Bignums, complex numbers, extendable arithmetic (no infix)
- Optional strict typing, assertions and fast, optimising compilers
- Built-in arrays, vectors, hash-tables and other data structures
- Object System (CLOS)
- Interactive environment[(a)]

[(a)] Here “interactive” is meant in the sense that the working system can be inspected and changed during its operation.

Lisp Present: Emacs Lisp

- Created by Richard M. Stallman
- Largely inspired by MacLisp
- Indefinite scope
- VM-like with the GNU/Emacs core as OS binding

Notable omissions

Here is a taste of some things we didn't mention from various Lisps:

- compilers
- CLOS, Meta-object protocol
- Scheme continuations
- exact, rich arithmetic
- conditions and restarts
- programmable parser/serializers

Bouquets and Brickbats

> Greenspun's Tenth Rule of Programming: any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

– Philip Greenspun

> Lisp is a beautiful language, but all the programs written in Lisp turn out horribly ugly.

– Larry Wall, OSDC.org.il 2006 (paraphrased)

Bouquets and Brickbats II

> The greatest single programming language ever designed

– Alan Kay (designer of the Smalltalk language), On Lisp

> Plenty of brilliant programmers know lisp just fine and still choose other languages. Most of them, in fact.

– Joel Spolsky, Fogcreek software

Bibliography

References

[1] “Structure and Interpretation of Computer Programs” By Harold Abelson and Gerald Jay Sussman with Julie Sussman, Massachusetts Institute of Technology, 1996.

[2] “The Calculi of Lambda-Conversion” By A. Church, Princeton University Press, Princeton, N.J., 1941.

[3] “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I” By John McCarthy, Massachusetts Institute of Technology, Cambridge Mass. April 1960.

Bibliography II

References

[4] “Lisp 1.5 Programmer’s Manual” By John McCarthy et al. The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology. August 17, 1962.

[5] “On Lisp” By Paul Graham, Prentice Hall, 1993

[6] “The Evolution of Lisp” by Guy L. Steele Jr. and Richard P. Gabriel.

Bibliography III

References

[7] “DATA REPRESENTATION IN PDP-10 MACLISP” by Guy Lewis Steele Jr., AI Memo 420, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, September 1977.

[8] “Common LISP: The Language” By Guy L. Steele Jr., Digital Equipment Corporation Digital Press, 1984

[9] “The Function of FUNCTION in LISP” By Joel Moses, AI Project MAC Memo AI-199, June 1970