Common LISP

The Programmable Programming Language

Santiago Martin Zubieta Ortiz
Universidad EAFIT

“The programmable programming language”

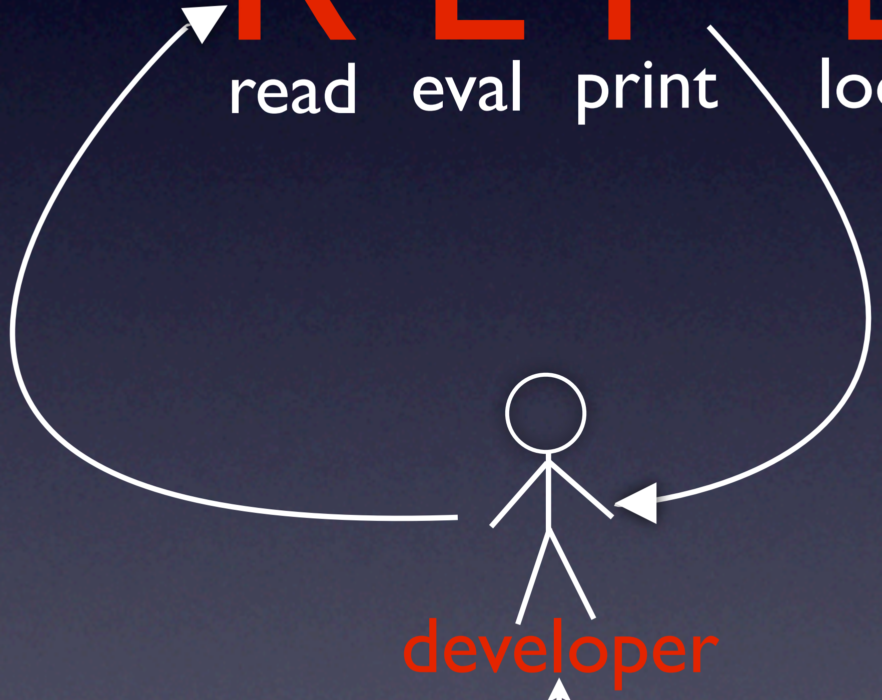
“You almost never find yourself wishing the language supported some feature that would make your program easier to write, because as you’ll see through, you can just add the feature yourself’

Exploratory programming

continually interact with the program while developing it

R E P L

read eval print loop



Dynamic Typing

less time convincing the compiler something should run and more time running it.

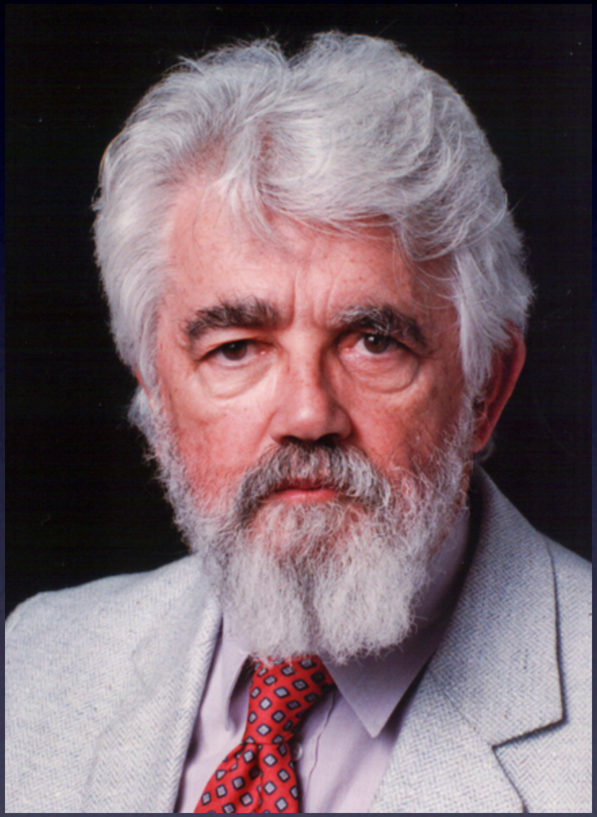
CLCS

Common Lisp Condition System allows to develop the error handling code on an upper layer of abstraction interactively

If you don't know exactly how your program is going to work when you first si
down to write it, Common Lisp provides several features to help you develop you
code incrementally and interactively.

Beginning

John McCarthy



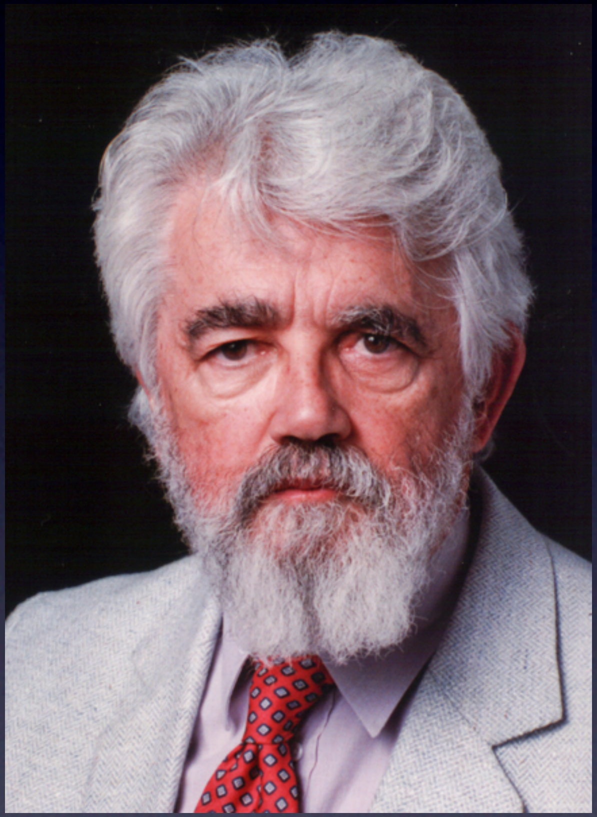
1927 - 2011

Common LISP is the modern descendant of the LISP language first conceived by John McCarthy in 1956. LISP was conceived for “Symbolic Data Processing” and derived its name from one of the things it was good at:

LISt Processing

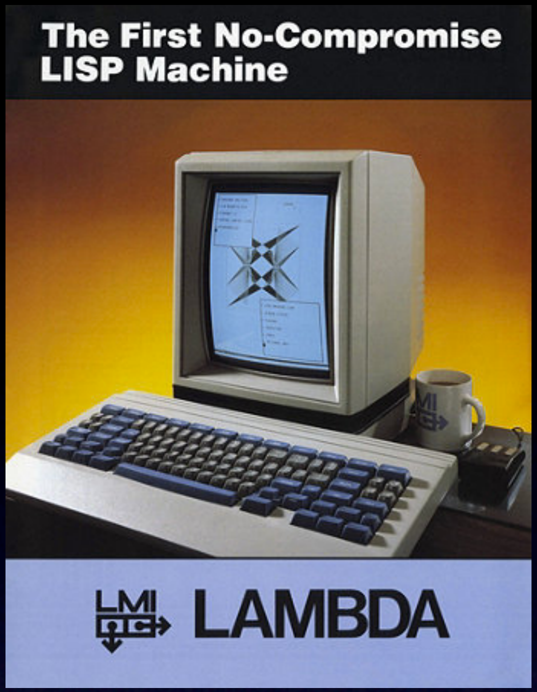
Beginning

John McCarthy



1927 - 2011

A.I. Researcher



LISP machines!

1980's A.I. Boom

Automated Theorem Proving

Planning and Scheduling

Computer Vision

Cold War, DARPA

Battlefield Simulations

Automated Planning

Natural Language Interfaces

Paradigms

LISP has roots in the Functional Programming Paradigm, albeit with time it has evolved into a multi-paradigm language.

“Whatever paradigm comes up next, it’s extremely likely that Common LISP will be able to absorb it without requiring any changes to the core language”

This shows how it isn’t really about LISP having a certain paradigm resulting from integration or creation, but how its ‘programability’ allows developers to configure it to support new paradigms easily.

We could be talking about the ultimate programming paradigm, a ‘Language-Oriented’ Paradigm, where no matter what paradigm comes up, it can be implemented!

Evolution, Paradigms

Common LISP has expanded from its functional core by getting powerful facilities for Object Oriented Programming using the Common Lisp Object System (CLOS) this one started as a library! For some people, Aspect Oriented Programming seems to be the next thing, and Common LISP already has a ongoing library for it!

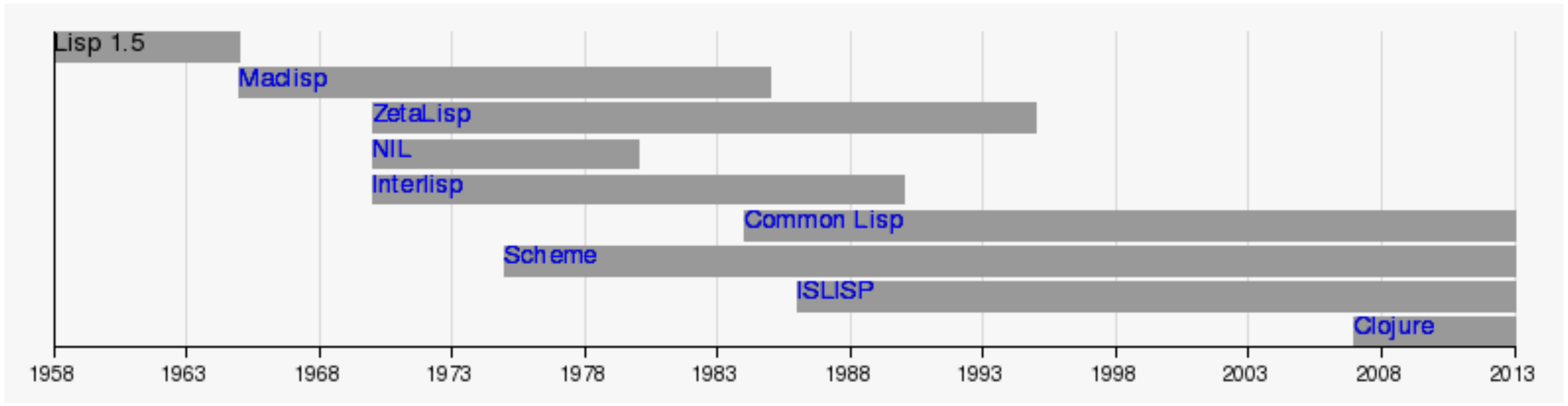
It has also got a fine array of modern data types, which coupled with dynamic typing give flexibility and spend less time convincing the compiler a program should run and have more time actually running it and working

And with the Common Lisp Condition System, it gains a whole new level of flexibility missing from the exception systems in many modern languages.

Lisp introduced the concept of automatic garbage collection, in which the system walks the heap looking for unused memory. Most of the modern sophisticated garbage collection algorithms such as generational garbage collection were developed for Lisp.

It has come a
long way!

Dialects of LISP



Functions

Notation

Polish Notation + 3 4

List

(+ 3 4)

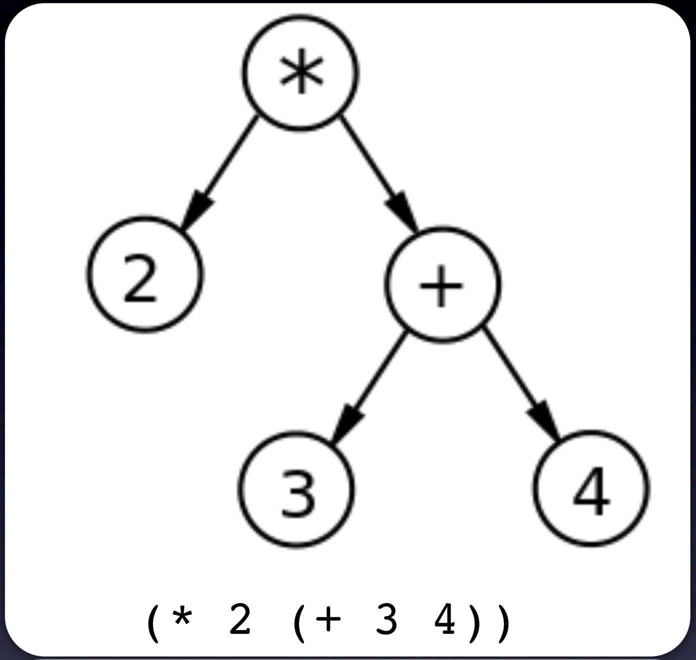
function arguments

These lists are self evaluating objects, so when given to the REPL read by R, the E will immediately evaluate it, and then P prints the returned value

10 even just a number is a self-evaluating object. also strings, etc.

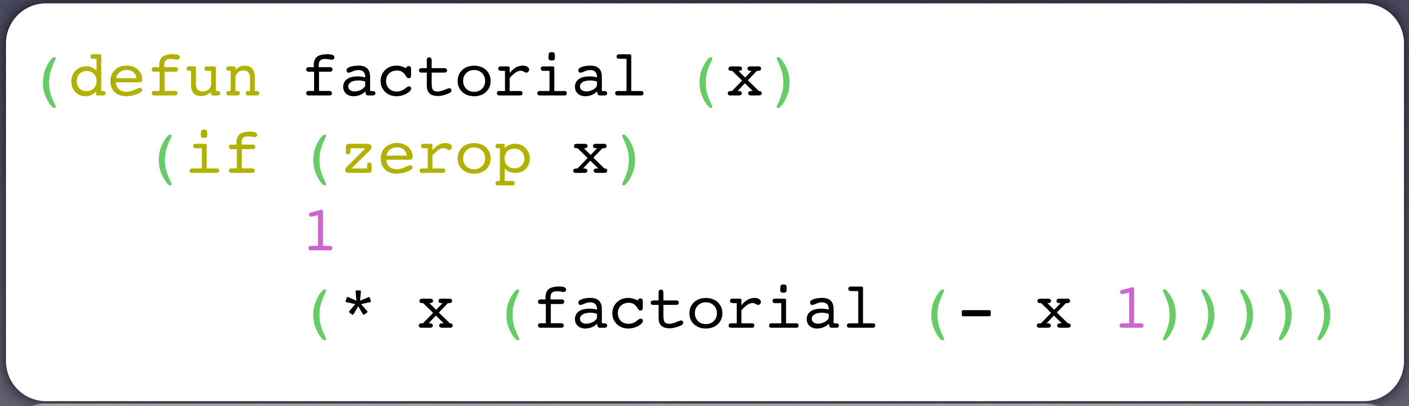
S-Expressions

Symbolic Expressions



Nested List, tree structured data

- An atom
- A expression of the form (x . y) where x and y are S-Expressions



```
(defun factorial (x)
   (if (zerop x)
       1
       (* x (factorial (- x 1)))))
```

Data Types

Numbers	Integers, Rationals, Complex numbers of arbitrary precision
Characters	Represent printed glyphs such as letters or text formatting operations. Strings are one-dimensional arrays of characters. Common Lisp provides for a rich character set, including ways to represent characters of various type styles.
Symbols	Atomic Symbols are named data objects. Lisp provides machinery for locating a symbol object, given its name (in the form of a string). Symbols have property lists, which in effect allow symbols to be treated as record structures with an extensible set of named components, each of which may be any Lisp object. Symbols also serve to name functions and variables within programs.
Lists	Are sequences represented in the form of linked cells called conses. The symbol **nil** is the empty list. All other lists are built recursively by adding a new element to the front of an existing list. This is done by creating a new cons, which is an object having two components called the car and the cdr. The car may hold anything, and the cdr is made to point to the previously existing list.

http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html

Data Types

Arrays	Dimensioned collections of objects that may contain any kind of data-type inside, though some are specialized for efficiency and allow only one type.
Hash Tables	Hash tables provide an efficient way of mapping any Lisp object (a key) to an associated object.
Structures	User-defined record structures, objects that have named components. The defstruct facility is used to define new structure types. Some Common Lisp implementations may choose to implement certain system-supplied data types, such as bignums, streams, hashtables, as structures, but this fact will be invisible to the user.
Streams	Represent sources or sinks of data, typically characters or bytes. They are used to perform I/O, as well as for internal purposes such as parsing strings.

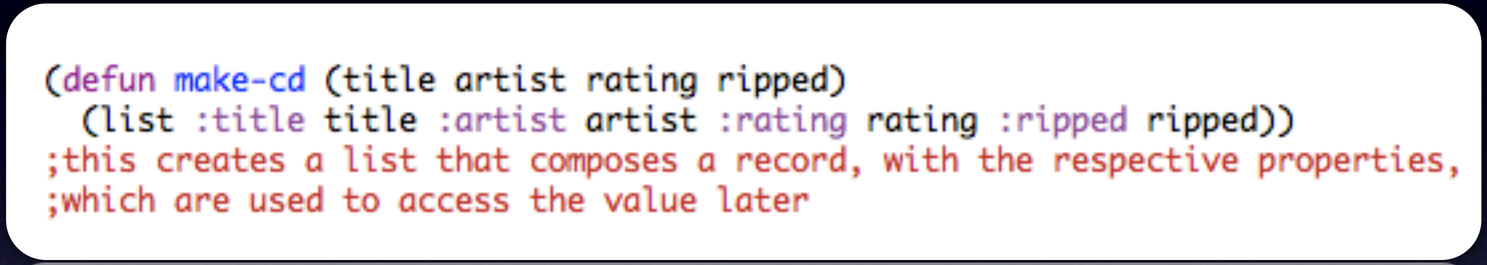
http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node15.html

Lists

Creating a list

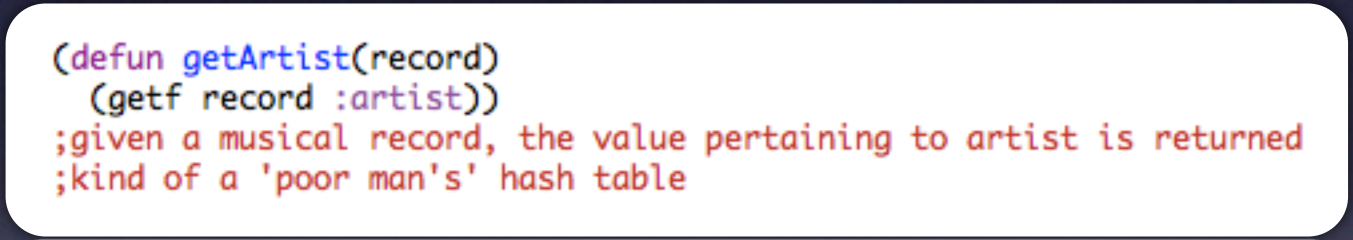
(list V_1 V_2 V_3.. V_n)   ‘(V_1 V_2 V_3.. V_n)

Use of properties



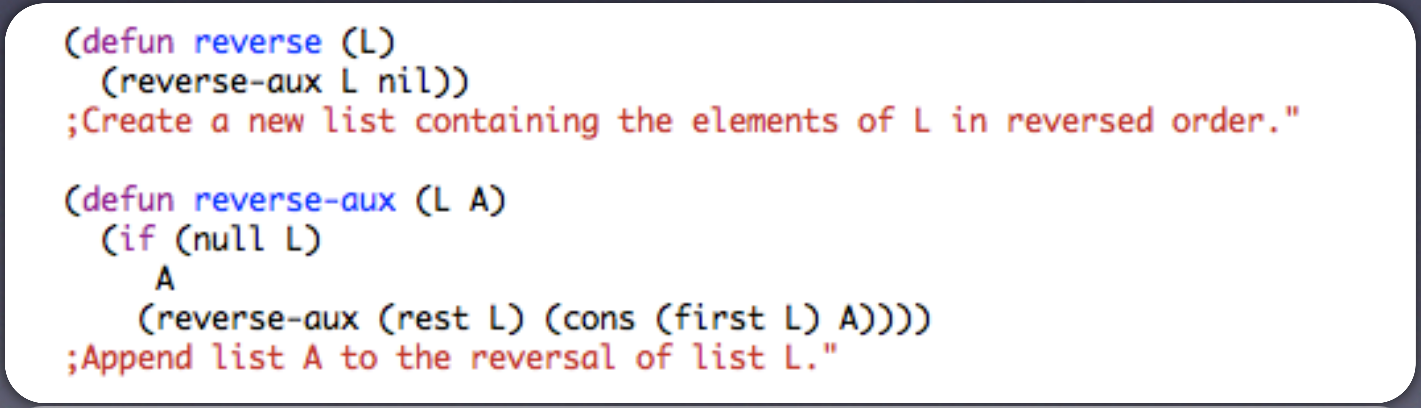
```
(defun make-cd (title artist rating ripped)
  (list :title title :artist artist :rating rating :ripped ripped))
;this creates a list that composes a record, with the respective properties,
;which are used to access the value later
```

Accessing a value of a p-list



```
(defun getArtist(record)
  (getf record :artist))
;given a musical record, the value pertaining to artist is returned
;kind of a 'poor man's' hash table
```

Common list operation example



```
(defun reverse (L)
  (reverse-aux L nil))
;Create a new list containing the elements of L in reversed order."

(defun reverse-aux (L A)
  (if (null L)
      A
    (reverse-aux (rest L) (cons (first L) A))))
;Append list A to the reversal of list L."
```

Lists

Use of a global variable

```
(defvar *db* nil)
;*db* is now a global variable initialized as nil
```

Inserting elements into a list

```
(defun add-record (cd)
  (push cd *db*))
;a certain thing cd, which will be a list later, is pushed into *db*
```

Printing a list contents with formatting

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a:~10t~a~%~}~%" cd)))

;~a aesthetic directive, consume one orgument, output human readable without the : in prop-
;-erties and strings without quotation marks
;~10t tabulation, enough spaces until the 10th column
;~{ and ~} means that next argument consumed is a list, and format will iterate inside it
;~% makes format emit a new line

(defun dump-db2 ()
  (format t "~{~{~a:~10t~a~%~}~%~}" *db*))
;since *db* is a list, it could also be iterated that way!
```

Vectors

Notation

```
(vector)       ==> #()
(vector 1)     ==> #(1)
(vector 1 2)   ==> #(1 2)
```

Name	Required Arguments	Returns
COUNT	Item and sequence	Number of times item appears in sequence
FIND	Item and sequence	Item or **NIL**
POSITION	Item and sequence	Index into sequence or **NIL**
REMOVE	Item and sequence	Sequence with instances of item removed
SUBSTITUTE	New item, item, and sequence	Sequence with instances of item replaced with new item

Examples

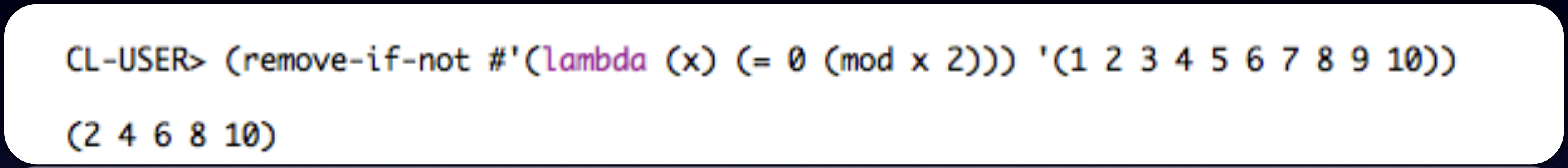
```
(count 1 #(1 2 1 2 3 1 2 3 4))           ==> 3
(remove 1 #(1 2 1 2 3 1 2 3 4))          ==> #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4))          ==> (2 2 3 2 3 4)
(remove #\a "foobarbaz")                 ==> "foobrbz"
(substitute 10 1 #(1 2 1 2 3 1 2 3 4))   ==> #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4))   ==> (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz")         ==> "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4))            ==> 1
(find 10 #(1 2 1 2 3 1 2 3 4))           ==> NIL
(position 1 #(1 2 1 2 3 1 2 3 4))        ==> 0
```

High-order Function Variants

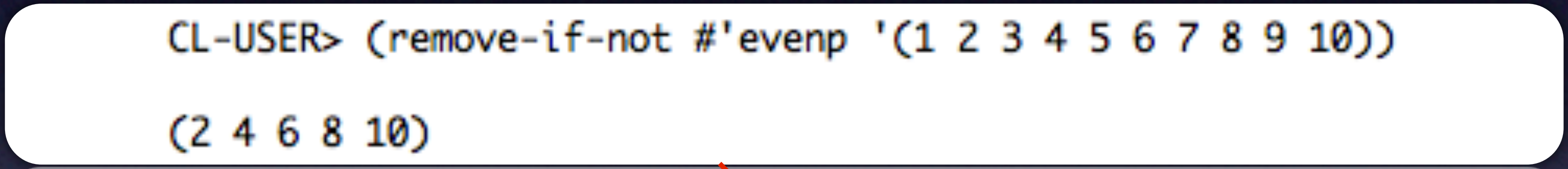
```
(count-if #'evenp #(1 2 3 4 5))            ==> 2

(count-if-not #'evenp #(1 2 3 4 5))        ==> 3

(position-if #'digit-char-p "abcd0001")    ==> 4

(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom")) ==> #("foo" "foom")
```

Functions

Anonymous functions



```
CL-USER> (remove-if-not #'(lambda (x) (= 0 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))

(2 4 6 8 10)
```



```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))

(2 4 6 8 10)
```

remove-if-not:

take a predicate, a list and return a new list containing the elements that match the predicate.

In this case, the predicate is the function **evenp**, which returns true if its argument is an even number. The notation #' is shorthand for "Get me the function with the following name." Without the #', Lisp would treat **evenp** as the name of a variable and look up the value of the variable, not the function.

Macros

Not the macros you're thinking of!

Regular languages have their cores extended via libraries, written almost entirely in the very same language, while LISP macros give a more expressive way, where each macro can have its own syntax, determining how the S-expressions that are passed are turned into LISP forms.

With macros its possible to build syntax-control constructs such as **WHEN** **DO-LIST** and **LOOP** or definitional forms such as **DEFUN** and **DEFPARAMETER** without hardwiring them into the core.

A macro is a function that takes s-expressions as arguments and returns a Lisp form that's then evaluated in place of the macro form

Macros

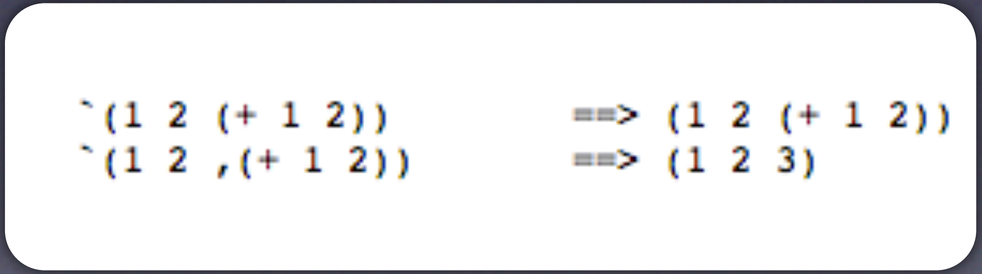
Not the macros you're thinking of!

The evaluation of a macro form proceeds in two phases: First, the elements of the macro form are passed, unevaluated, to the macro function. Second, the form returned by the macro function--called its expansion--is evaluated according to the normal evaluation rules.

Since the evaluator doesn't evaluate the elements of the macro form before passing them to the macro function, they don't need to be well-formed Lisp forms. Each macro assigns a meaning to the s-expressions in the macro form by virtue of how it uses them to generate its expansion. In other words, each macro defines its own local syntax.

Important for macro construction!

A back quote (`) before an expression stops evaluation just like a forward quote. However, in a back-quoted expression, any subexpression that's preceded by a comma is evaluated. Notice the effect of the comma in the second expression:



```
`(1 2 (+ 1 2))         ==> (1 2 (+ 1 2))
`(1 2 ,(+ 1 2))        ==> (1 2 3)
```

Macros

Regular if-then-else form

```
(if (spam-p current-message)
    (file-in-spam-folder current-message)
    (update-spam-database current-message))
```

Only allows one lisp form at if and else

This ubiquitous structure was invented by McCarthy for use in LISP! (as cond)

More than one form, 'not bad'

```
(if (spam-p current-message)
    (progn
      (file-in-spam-folder current-message)
      (update-spam-database current-message)))
```

This gives a functional program procedural facilities

'when' standard macro

```
(when (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

Creating the 'when' and 'unless' macro

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

```
(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

Other Macros

cond

Ugly!

```
(if a
    (do-x)
    (if b
       (do-y)
       (do-z)))
```

Nice!

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

This ubiquitous structure was invented by McCarthy for use in LISP!

dotimes

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

```
(dotimes (var count-form)
  body-form*)
```

do

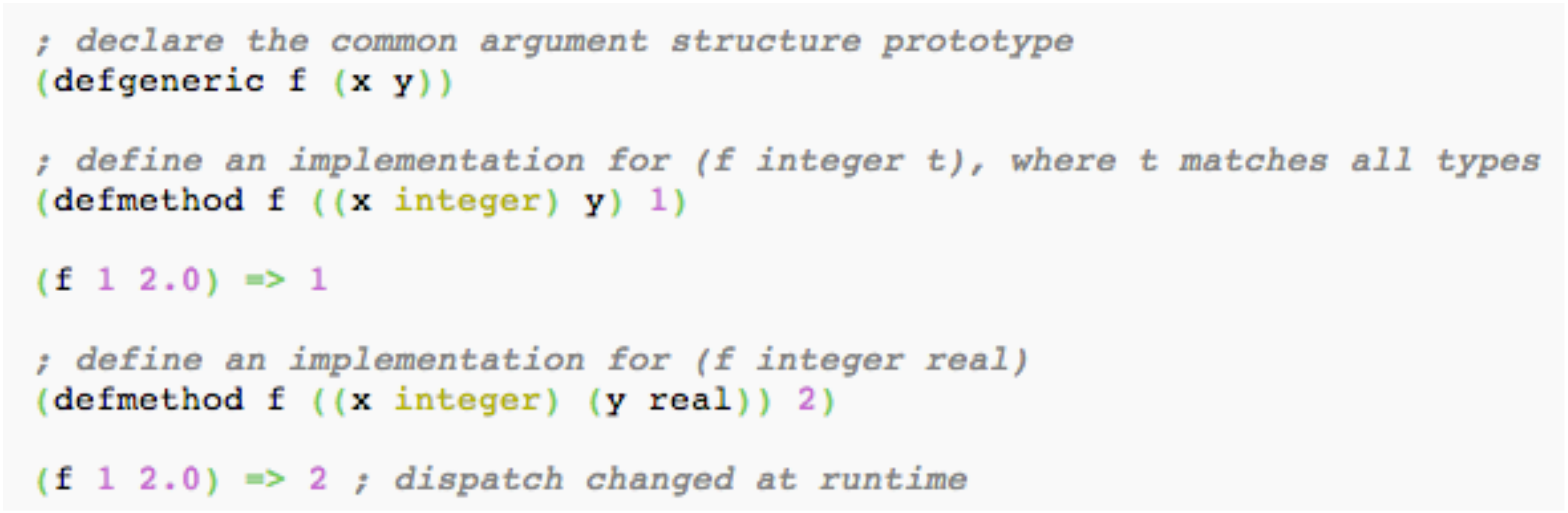
```
(var init-form step-form)
```

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

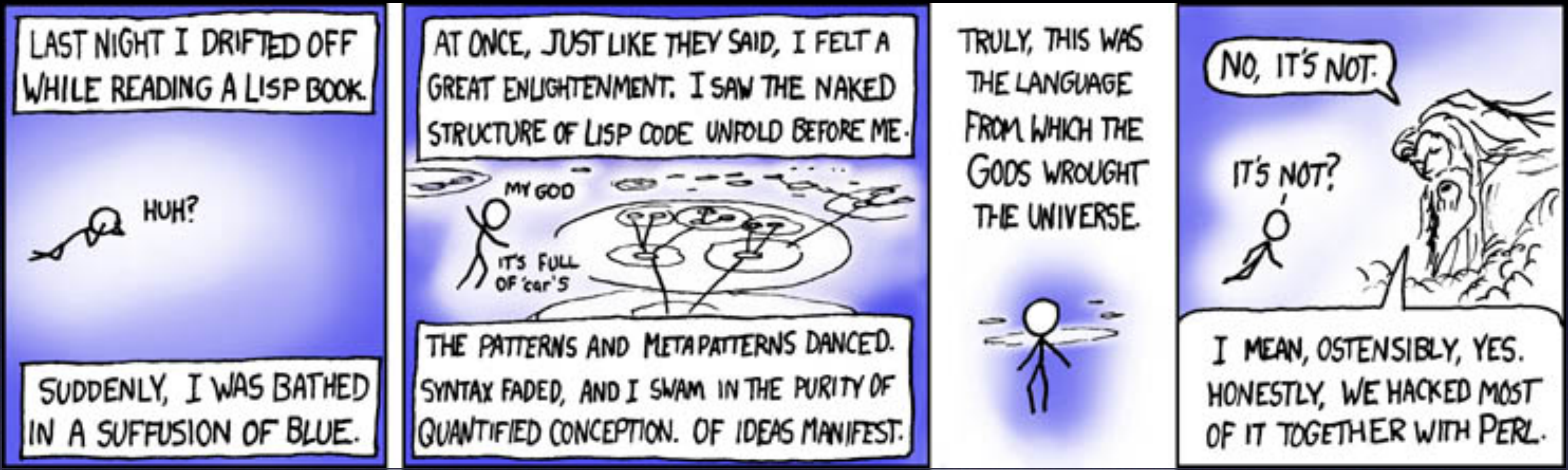
```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

Eleventh fibonacci number

Object System



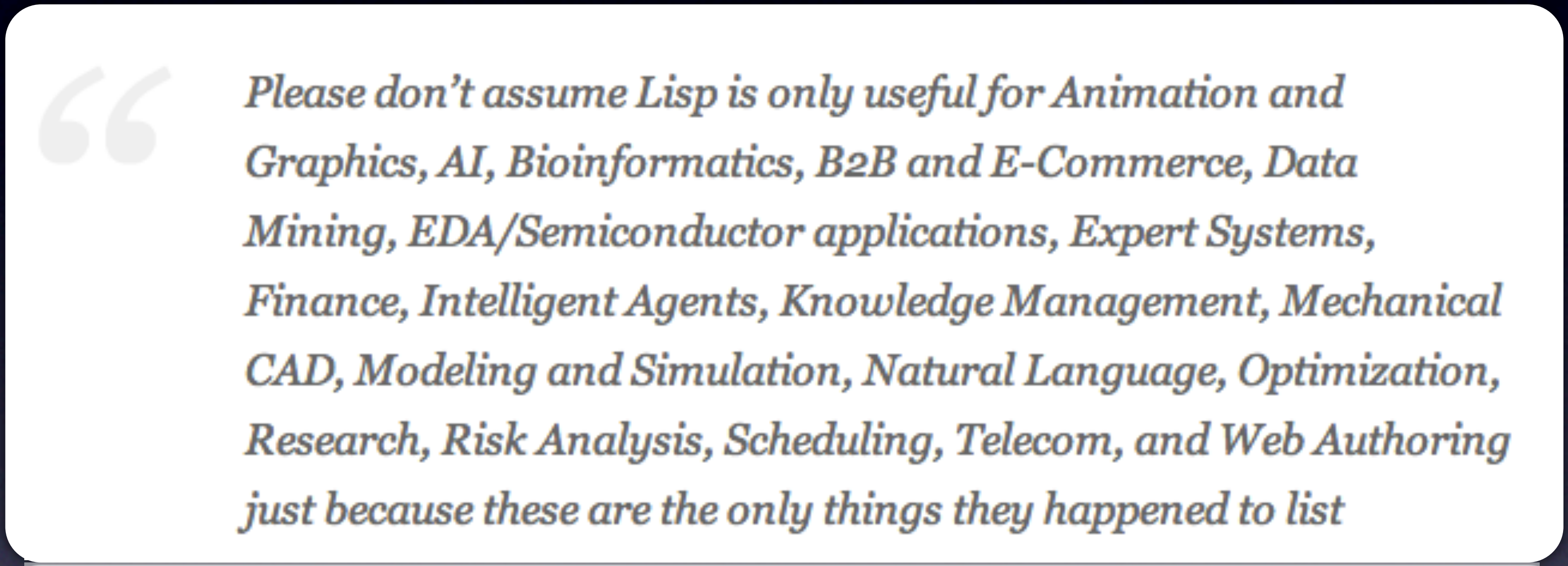
```lisp
; declare the common argument structure prototype
(defgeneric f (x y))

; define an implementation for (f integer t), where t matches all types
(defmethod f ((x integer) y) 1)

(f 1 2.0) => 1

; define an implementation for (f integer real)
(defmethod f ((x integer) (y real)) 2)

(f 1 2.0) => 2 ; dispatch changed at runtime
```


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

"Lisp is the ultimate power tool. The language can be extended by the programmer in almost any way it sees fit, without having to wait for Lisp 2.0 or 3.0. It appears to be the ultimate road to infinite programmer expressivity."

http://www.stateofcode.com/2011/06/lisp-the-programmable-programming-language-with-manuel-simoni/

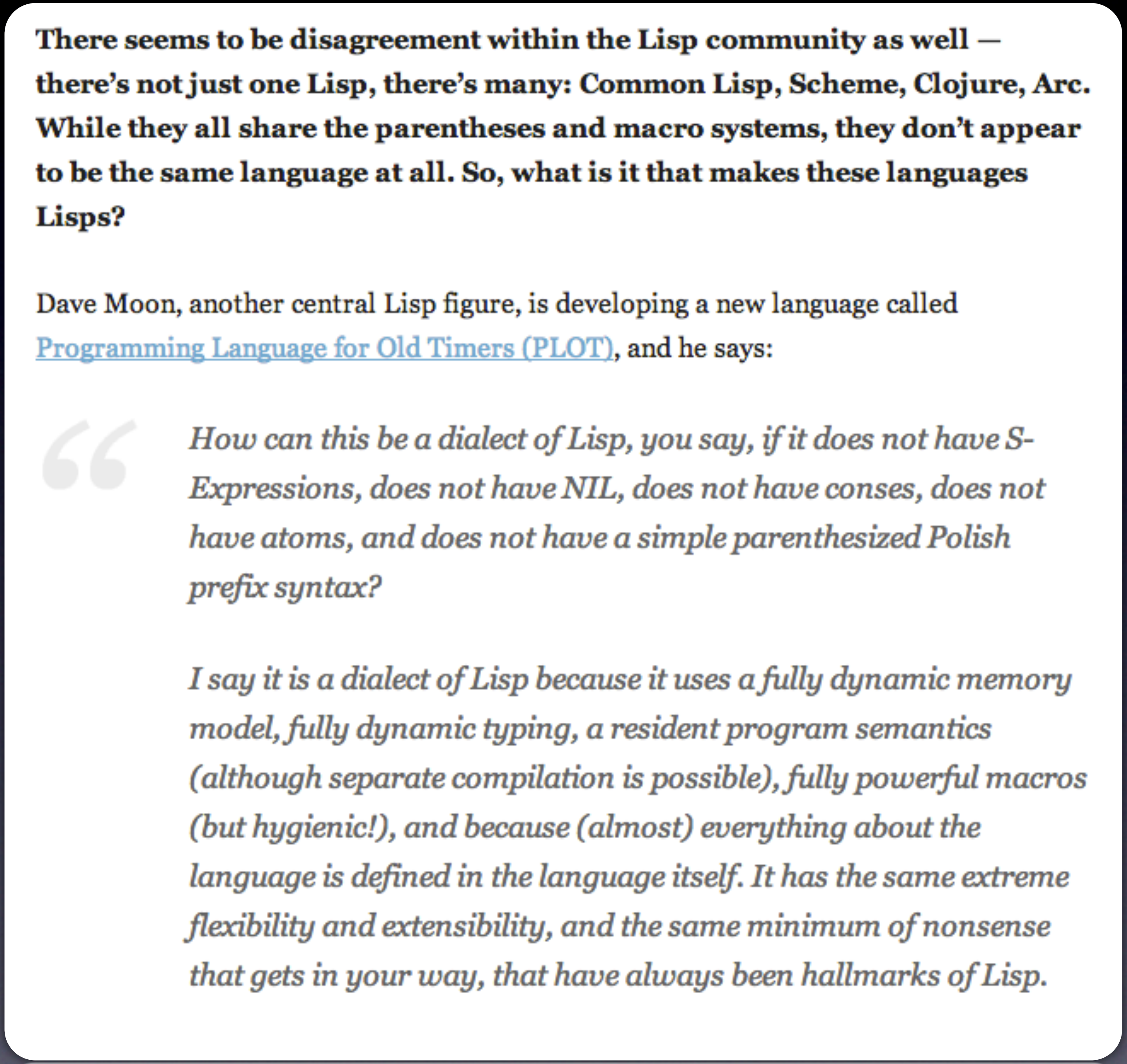
Applications



"Please don't assume Lisp is only useful for Animation and Graphics, AI, Bioinformatics, B2B and E-Commerce, Data Mining, EDA/Semiconductor applications, Expert Systems, Finance, Intelligent Agents, Knowledge Management, Mechanical CAD, Modeling and Simulation, Natural Language, Optimization, Research, Risk Analysis, Scheduling, Telecom, and Web Authoring just because these are the only things they happened to list

http://www.stateofcode.com/2011/06/lisp-the-programmable-programming-language-with-manuel-simoni/

Dialects of LISP



There seems to be disagreement within the Lisp community as well — there's not just one Lisp, there's many: Common Lisp, Scheme, Clojure, Arc. While they all share the parentheses and macro systems, they don't appear to be the same language at all. So, what is it that makes these languages Lisps?

Dave Moon, another central Lisp figure, is developing a new language called Programming Language for Old Timers (PLOT), and he says:

> *How can this be a dialect of Lisp, you say, if it does not have S-Expressions, does not have NIL, does not have conses, does not have atoms, and does not have a simple parenthesized Polish prefix syntax?*
>
> *I say it is a dialect of Lisp because it uses a fully dynamic memory model, fully dynamic typing, a resident program semantics (although separate compilation is possible), fully powerful macros (but hygienic!), and because (almost) everything about the language is defined in the language itself. It has the same extreme flexibility and extensibility, and the same minimum of nonsense that gets in your way, that have always been hallmarks of Lisp.*

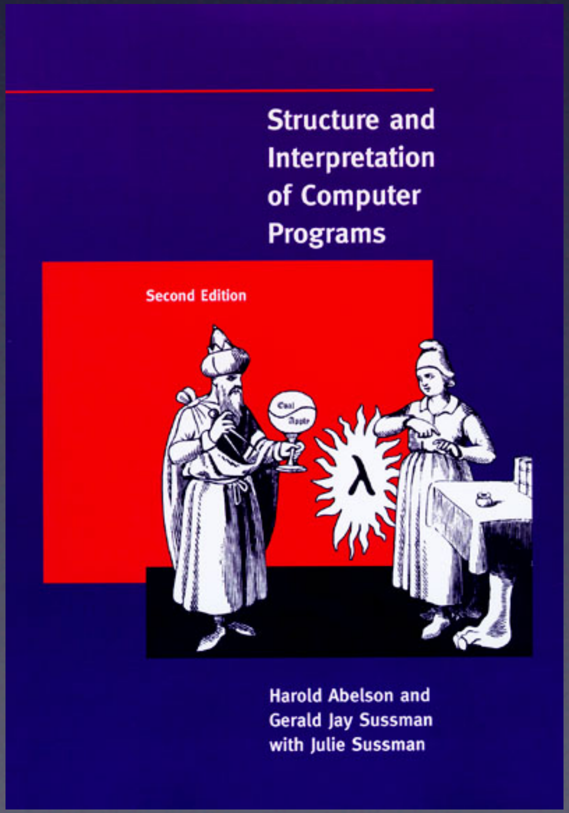
http://www.stateofcode.com/2011/06/lisp-the-programmable-programming-language-with-manuel-simoni/

Aspirations

I see in LISP a language that I first thought was ancient and limited to environments like EMACS. Then I realized how flexible it is, and how much applications it has (even Graphic programs like AutoCAD has its own LISP dialect), with plenty of dialects that sometimes they strife away from the main language, but conserving the power and flexibility that has made it so powerful during almost 50 years. I find in it a language that can help me not only implement ideas from different paradigms of thinking, but also delving into more 'hardcore' stuff like...

Structure and Interpretation of Computer Programs (SICP) is a textbook published in 1984 about general computer programmingconcepts from MIT Press written by Massachusetts Institute of Technology (MIT) professors Harold Abelson and Gerald Jay Sussman, with Julie Sussman. It was formerly used as the textbook of MIT introductory programming class and at other schools.

Using a dialect of the Lisp programming language known as Scheme, the book explains core computer science concepts, includingabstraction, recursion, interpreters and metalinguistic abstraction, and teaches modular programming.


Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

http://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs

Bibliography

Practical Common Lisp. Peter Seibel,
http://www.gigamonkeys.com/book/
2003-2009