

*INF4820: Algorithms for  
Artificial Intelligence and  
Natural Language Processing*

Common Lisp Fundamentals

Stephan Oepen & Murhaf Fares

Language Technology Group (LTG)

September 1, 2016



- ▶ **Since the 1950s:** Chatbots, theorem proving, blocks world, expert and dialogue systems, game playing, ...
  - ▶ Moving target: Whatever requires 'intelligent' decisions, but seems out of reach, technologically, at the time?
  - ▶ **Recently:** Conversational user interfaces, self-driving cars, talking robots, AlphaGo.
  - ▶ But also (**fuzzily**) business intelligence, (big) data analytics, ...
- Toolkit of ('clever') methods for representation and problem solving.





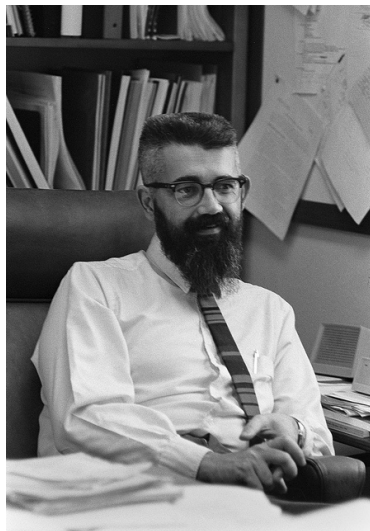
# Lisp

Eric S. Raymond (2001), *How to Become a Hacker*:

*Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.*

- ▶ High-level and efficient language with especially strong support for **symbolic** and **functional** programming.
- ▶ Rich language: multitude of built-in data types and operations.
- ▶ Easy to learn:
  - ▶ extremely simple syntax,
  - ▶ straightforward semantics.
- ▶ ANSI-standardized and stable.
- ▶ Incremental and interactive development.

- ▶ Conceived in the late 1950s by John McCarthy—one of the founding fathers of AI.
- ▶ Originally intended as a mathematical formalism.
- ▶ A family of high-level languages.
- ▶ Several dialects, e.g. Scheme, Clojure, Emacs Lisp, and **Common Lisp**.
- ▶ Although a multi-paradigm language, functional style prevalent.





- ▶ Testing a few expressions at the **REPL**;
- ▶ the **read-eval-print** loop.
- ▶ (= the interactive Lisp-environment)
- ▶ **'?**' represents the REPL prompt and **'→'** what an expression evaluates to.
- ▶ Atomic data types like numbers, booleans, and strings are **self evaluating**.
- ▶ Symbols evaluate to whatever value they are bound to.

## Examples

```
? "this is a string"
→ "this is a string"

? 42
→ 42

? t
→ t

? nil
→ nil

? pi
→ 3.141592653589793d0

? foo
→ error; unbound
```



- ▶ Lisp manipulates so-called *symbolic expressions*.
- ▶ AKA s-expressions or **sexps**.
- ▶ Two fundamental types of sexps;
  1. **atoms** (e.g., `nil`, `t`, numbers, strings, symbols)
  2. **lists** containing other sexps.
- ▶ Sexps are used to represent *both* **data** and **code**.

- ▶ “Parenthesized prefix notation”
- ▶ First element (prefix) = **operator** (i.e. the procedure or function).
- ▶ The rest of the list is the **operands** (i.e. the arguments or parameters).
- ▶ Use nesting (of lists) to build compound expressions.
- ▶ Expressions can span multiple lines; indentation for readability.

## Examples

? (+ 1 2)  
→ 3

? (+ 1 2 10 7 5)  
→ 25

? (/ (+ 10 20) 2)  
→ 15

? (\* (+ 42 58)  
      (- (/ 8 2) 2))  
→ 200





```
? (expt (- 8 4) 2)
```

```
→ 16
```

- ▶ You now know (almost) all there is to know about (the rules of) CL.
- ▶ The first element of a list names a **function** that is invoked with the **values** of all remaining elements as its arguments.
- ▶ A few exceptions, called **special forms**, with their own evaluation rules.

- ▶ The special form **defun** associates a function definition with a symbol:

## General form

```
(defun name (parameter1 ... parametern) body)
```

## Example

```
? (defun average (x y)  
    (/ (+ x y) 2))
```

```
? (average 10 20)
```

```
→15
```



- ▶ Classic example: the **factorial** function.
- ▶ A **recursive** procedure; calls itself, directly or indirectly.
- ▶ May seem circular, but is well-defined as long as there's a **base case** terminating the recursion.
- ▶ For comparison: a non-recursive implementation (in Python).

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{if } n > 0 \end{cases}$$

```
(defun ! (n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))
```

```
def fac(n):
    r = 1
    while (n > 0):
        r = r * n
        n = n - 1
    return r
```

# A Special Case of Recursion: Tail Recursion



- ▶ A more efficient way to define  $n!$  recursively.
- ▶ Use a helper procedure with an **accumulator** variable to collect the product along the way.
- ▶ The recursive call is in **tail position**;

```
(defun ! (n)
  (!-aux 1 1 n))

(defun !-aux (r i n)
  (if (> i n)
      r
      (!-aux (* i r)
              (+ i 1)
              n))))
```

- ▶ no work remains to be done in the calling function.
- ▶ Once we reach the base case, the return value is ready.
- ▶ Most CL compilers do *tail call optimization* (TCO), so that the recursion is executed as an iterative loop.
- ▶ (The next lecture will cover CL's built-in loop construct.)

## Recursive

```
(defun ! (n)
  (if (= n 0)
      1
      (* n (! (- n 1)))))
```

```
? (! 7)
⇒ (* 7 (! 6))
⇒ (* 7 (* 6 (! 5)))
⇒ (* 7 (* 6 (* 5 (! 4))))
⇒ (* 7 (* 6 (* 5 (* 4 (! 3)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (! 2)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (! 1)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 2)))))
⇒ (* 7 (* 6 (* 5 (* 4 6)))))
⇒ (* 7 (* 6 (* 5 24)))
⇒ (* 7 (* 6 120))
⇒ (* 7 720)
→ 5040
```

## Tail-Recursive

```
(defun ! (n)
  (!-aux 1 1 n))

(defun !-aux (r i n)
  (if (> i n)
      r
      (!-aux (* r i)
              (+ i 1)
              n)))
```

```
? (! 7)
⇒ (!-aux 1 1 7)
⇒ (!-aux 1 2 7)
⇒ (!-aux 2 3 7)
⇒ (!-aux 6 4 7)
⇒ (!-aux 24 5 7)
⇒ (!-aux 120 6 7)
⇒ (!-aux 720 7 7)
⇒ (!-aux 5040 8 7)
→ 5040
```

# The quote Operator



- ▶ A *special form* making expressions self-evaluating.
- ▶ The **quote** operator (or simply **'**) suppresses evaluation.

```
? pi → 3.141592653589793d0
```

```
? (quote pi) → pi
```

```
? 'pi → pi
```

```
? foobar → error; unbound variable
```

```
? 'foobar → foobar
```

```
? (* 2 pi) → 6.283185307179586d0
```

```
? '(* 2 pi) → (* 2 pi)
```

```
? () → error; missing procedure
```

```
? '() → ()
```



- ▶ We've mentioned how sexps are used to represent *both* **data** and **code**.
- ▶ Note the double role of lists:
- ▶ Lists are function calls:

```
? (* 10 (+ 2 3)) → 50
```

```
? (bar 1 2) → error; function bar undefined
```

- ▶ But, lists can also be **data**:

```
? '(foo bar) → (foo bar)
```

```
? (list 'foo 'bar) → (foo bar)
```

- **cons** builds up new lists; **first** and **rest** destructure them.

```
? (cons 1 (cons 2 (cons 3 nil))) → (1 2 3)
```

```
? (cons 0 '(1 2 3)) → (0 1 2 3)
```

```
? (first '(1 2 3)) → 1
```

```
? (rest '(1 2 3)) → (2 3)
```

```
? (first (rest '(1 2 3))) → 2
```

```
? (rest (rest (rest '(1 2 3)))) → nil
```

- Many additional list operations (derivable from the above), e.g.

```
? (list 1 2 3) → (1 2 3)
```

```
? (append '(1 2) '(3) '(4 5 6)) → (1 2 3 4 5 6)
```

```
? (length '(1 2 3)) → 3
```

```
? (reverse '(1 2 3)) → (3 2 1)
```

```
? (nth 2 '(1 2 3)) → 3
```

```
? (last '(1 2 3)) → (3)
```

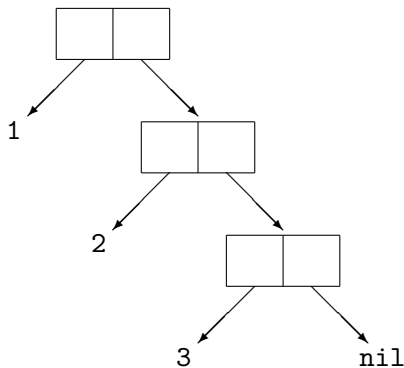
Wait, why not 3?



# Lists are Really Chained 'cons' Cells

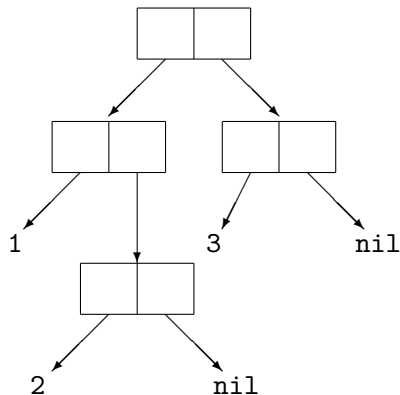


(1 2 3)



(cons 1 (cons 2 (cons 3 nil)))

((1 2) 3)



(cons (cons 1 (cons 2 nil)) (cons 3 nil))

# Assigning Values: 'Generalized Variables'



- ▶ **defparameter** declares a 'global variable' and assigns a value:

```
? (defparameter *foo* 42) → *F00*
```

```
? *foo* → 42
```

- ▶ **setf** provides a uniform way of assigning values to variables.
- ▶ General form:

```
(setf place value)
```

- ▶ ...where *place* can either be a variable named by a symbol or some other storage location:

```
? (setf *foo* (+ *foo* 1))
```

```
? *foo* → 43
```

```
? (setf *foo* '(2 2 3))
```

```
? (setf (first *foo*) 1)
```

```
? *foo* → (1 2 3)
```



Example	Type of <i>x</i>	Effect
<code>(<b>incf</b> x y)</code>	number	<code>(setf x (+ x y))</code>
<code>(<b>incf</b> x)</code>	number	<code>(incf x 1)</code>
<code>(<b>decf</b> x y)</code>	number	<code>(setf x (- x y))</code>
<code>(<b>decf</b> x)</code>	number	<code>(decf x 1)</code>
<code>(<b>push</b> y x)</code>	list	<code>(setf x (cons y x))</code>
<code>(<b>pop</b> x)</code>	list	<code>(let ((y (first x)))   (setf x (rest x)) y)</code>
<code>(<b>pushnew</b> y x)</code>	list	<code>(if (member y x)   x   (push y x))</code>

Shall we jointly write our own **push** and **pop**?

- ```
? (defparameter *foo* 42) → *FOO*
? (defparameter *bar* 100) → *BAR*
? (let ((*bar* 7)
        (baz 1))
    (+ baz *bar* *foo*))
→ 50
? *bar* → 100
? baz → error: unbound variable
```

- 20



- ▶ A *predicate* tests some condition.
- ▶ Evaluates to a boolean truth value:
  - ▶ `nil` (the empty list) means *false*.
  - ▶ Anything non-`nil` (including `t`) means *true*.

```
? (listp '(1 2 3)) → t
```

```
? (null (rest '(1 2 3))) → nil
```

```
? (evenp 2) → t
```

```
? (defparameter foo 42)
```

```
? (or (not (numberp foo))  
      (and (>= foo 0)  
           (<= foo 42))) → t
```

- ▶ Plethora of equality tests: `eq`, `eq1`, `equal`, and `equalp`.

# Equality for One and All



- ▶ `eq` tests object identity; not applicable for numbers or characters.
- ▶ `eq1` is like `eq`, but well-defined on numbers and characters.
- ▶ `equal` tests structural equivalence (recursively for lists and strings).
- ▶ `equalp` is like `equal` but insensitive to case and numeric type.

```
? (eq (list 1 2 3) '(1 2 3)) → nil
```

```
? (equal (list 1 2 3) '(1 2 3)) → t
```

```
? (eq 42 42) → ? [implementation-dependent]
```

```
? (eq1 42 42) → t
```

```
? (eq1 42 42.0) → nil
```

```
? (equalp 42 42.0) → t
```

```
? (equal "foo" "foo") → t
```

```
? (equalp "FOO" "foo") → t
```

- ▶ Also many type-specialized tests like `=`, `string=`, etc.

## Examples

```
? (defparameter foo 42)
```

```
? (if (numberp foo)
      "number"
      "something else")
→ "number"
```

```
? (cond ((< foo 3) "less")
        ((> foo 3) "more")
        (else "equal"))
→ "more"
```

## General Form

```
(if <predicate>
    <then clause>
    <else clause>)
```

```
(cond (<predicate1> <clause1>)
      (<predicate2> <clause2>)
      (<predicatei> <clausei>)
      (t <default clause>))
```



- ▶ **Symbols** can have values as **functions** and **variables** at the same time.
- ▶ **#'** (*sharp-quote*) gives us the function object bound to a symbol.

```
? (defun foo (x)
    (* x 1000))
```

```
? (defparameter foo 42) → 2
```

```
? (foo foo) → 42000
```

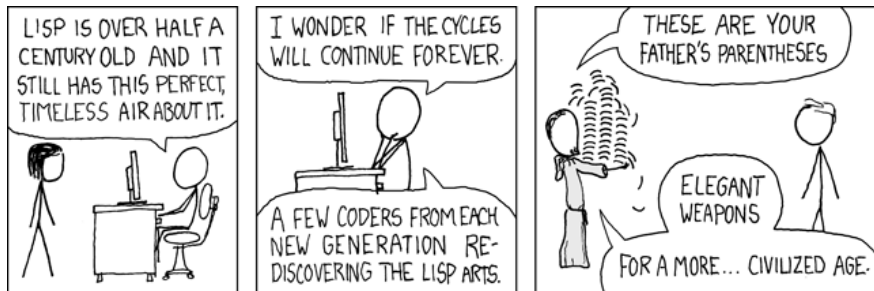
```
? foo → 42
```

```
? #'foo → #<Interpreted Function FOO>
```

```
? (funcall #'foo foo) → 42000
```

- ▶ **#'** and **funcall** (as well as **apply**) are useful when passing around functions as arguments.





<http://xkcd.com/297/>



## More Common Lisp.

- ▶ More on argument lists (optional arguments, keywords, defaults).
- ▶ More data types: Hash-tables, a-lists, arrays, sequences, and structures
- ▶ More higher-order functions.
- ▶ Iteration (loop) and mapping.