# — INF4820 —
# Algorithms for AI and NLP

*Common Lisp Essentials*

Erik Velldal & Stephan Oepen

Language Technology Group (LTG)

August 26, 2015

## Topic of the day

# Lisp

## Lisp

- 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.



## Basic common lisp in a couple of minutes

- 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
```

## A note on terminology

- 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.



## Function calls

- "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
```

## The syntax and semantics of CL

```
? (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.



## Creating our own functions

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

**General form**

(`defun` *name* (*parameter*$_1$ ... *parameter*$_n$) *body*)

**Example**

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

? (average 10 20)
→15
```

## Some other special forms

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

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

- Conditional evaluation with if and cond:

**Examples**

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

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

**General form**

```
(if ⟨predicate⟩
    ⟨then clause⟩
    ⟨else clause⟩)

 (cond (⟨predicate_1⟩ ⟨clause_1⟩)
       (⟨predicate_2⟩ ⟨clause_2⟩)
       (⟨predicate_i⟩ ⟨clause_i⟩)
       (t ⟨default clause⟩))
```



## The 'Hello World!' of functional programming

- 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 fac (n)
  (if (= n 0)
      1
      (* n (fac (- 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 fac (n)
  (fac-iter 1 1 n))

(defun fac-iter (prod count n)
  (if (> count n)
      prod
      (fac-iter (* count prod)
                (+ count 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*, so that the recursion is executed as an iterative loop.
- (The next lecture will cover CL's built-in `loop` construct.)



## Tracing the processes

### Recursive

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

```
? (fac 7)
⇒ (* 7 (fac 6))
⇒ (* 7 (* 6 (fac 5)))
⇒ (* 7 (* 6 (* 5 (fac 4))))
⇒ (* 7 (* 6 (* 5 (* 4 (fac 3)))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (fac 2))))))
⇒ (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (fac 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
```

### Iterative (tail recursive)

```
(defun fac (n)
  (fac-iter 1 1 n))

(defun fac-iter (prod count n)
  (if (> count n)
      prod
      (fac-iter (* count prod)
                (+ count 1)
                n)))
```

```
? (fac 7)
⇒ (fac-iter 1 1 7)
⇒ (fac-iter 1 2 7)
⇒ (fac-iter 2 3 7)
⇒ (fac-iter 6 4 7)
⇒ (fac-iter 24 5 7)
⇒ (fac-iter 120 6 7)
⇒ (fac-iter 720 7 7)
⇒ (fac-iter 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

? '() → ()
```



## Both code and data are s-expressions

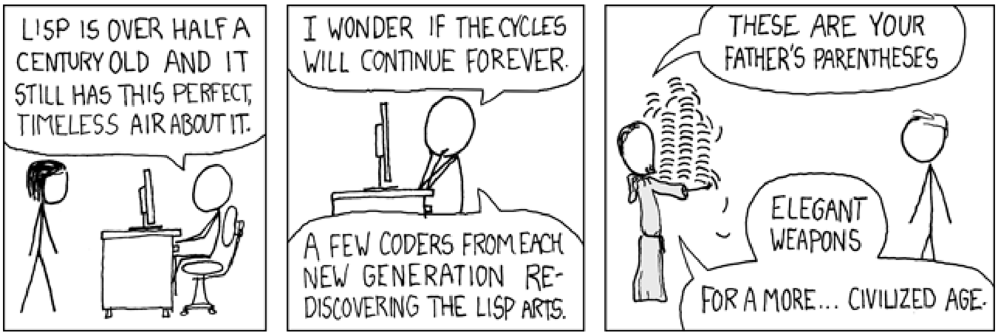
- We've mentioned how sexps are used to represent *both* data and code.
- Note the double role of lists:
- Lists are function calls;

```
? (* 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)
```

# Break




http://xkcd.com/297/

**Eric Raymond, *How to Become a Hacker*, 2001:**

*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 should never actually use Lisp itself a lot.*



# LISP = LISt Processing

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

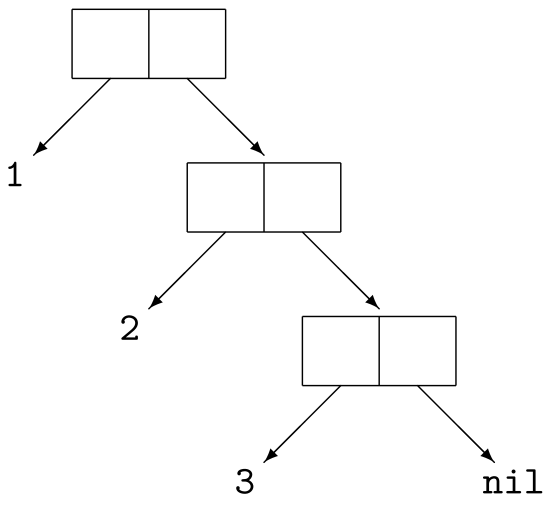
```
? (cons 1 (cons 2 (cons 3 nil))) → (1 2 3)
? (cons 0 '(1 2 3)) → (0 1 2 3)
? (first '(1 2 3)) → 1
? (rest '(1 2 3)) → (2 3)
? (first (rest '(1 2 3))) → 2
? (rest (rest (rest '(1 2 3)))) → nil
```

- 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?
```

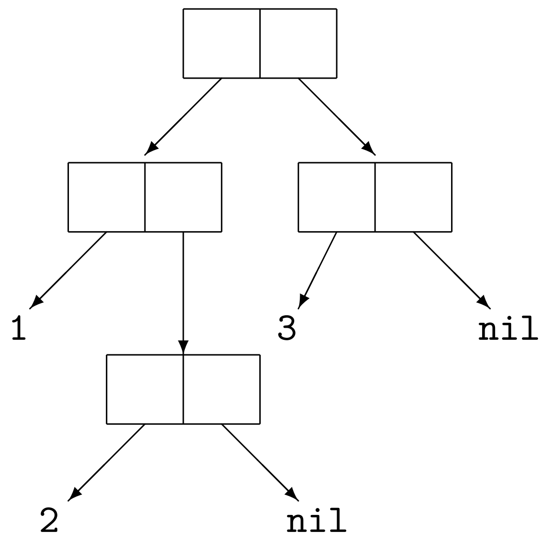
# Lists are really chained ‘cons cells’

`(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’

- `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:

```
? (defparameter *foo* 42)
? (setf *foo* (+ *foo* 1))
? *foo* → 43
? (setf *foo* '(2 2 3))
? (setf (first *foo*) 1)
? *foo* → (1 2 3)
```

## Some other macros for assignment

| Example | Type of `x` | Effect |
|---|---|---|
| `(incf x y)` | number | `(setf x (+ x y))` |
| `(incf x)` | number | `(incf x 1)` |
| `(decf x y)` | number | `(setf x (- x y))` |
| `(decf x)` | number | `(decf x 1)` |
| `(push y x)` | list | `(setf x (cons y x))` |
| `(pop x)` | list | `(let ((y (first x)))`<br>`  (setf x (rest x)) y)` |
| `(pushnew y x)` | list | `(if (member y x)`<br>`  x`<br>`  (push y x))` |



## Local variables

- Sometimes we want to store intermediate results.
- `let` and `let*` create temporary value bindings for symbols.

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

- Bindings valid only in the body of `let`.
- Previously existing bindings are *shadowed* within the lexical scope.
- `let*` is like `let` but binds *sequentially*.

## Predicates

- 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`, `eql`, `equal`, and `equalp`.



## Equality for one and all

- `eq` tests object identity; it is not useful for numbers or characters.
- `eql` is like `eq`, but well-defined on numbers and characters.
- `equal` tests structural equivalence
- `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]

? (eql 42 42) → t

? (eql 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.

## Rewind: A note on symbol semantics

- 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.



## Higher-order functions

- Functions that accept functions as arguments or return values.
- Functions in Lisp are first-class objects.
  - Can be created at run-time, passed as arguments, returned as values, stored in variables... just like any other data type.

```
? (defun filter (list test)
   (cond ((null list) nil)
         ((funcall test (first list))
          (cons (first list)
                (filter (rest list) test)))
         (t (filter (rest list) test))))

? (defparameter foo '(11 22 33 44 55))

? (filter foo #'evenp)
→ (22 44)
```

## Anonymous functions

- We can also pass function arguments without first binding them to a name, using lambda expressions: `(lambda (`*parameters*`)` *body*`)`
- A function definition without the `defun` and *symbol* part.

```
? (filter foo
          #'(lambda (x)
              (and (> x 20)
                   (< x 50))))
→ (22 33 44)
```

- Typically used for ad-hoc functions that are only locally relevant and simple enough to be expressed inline.
- Or, when constructing functions as return values.



## Returning functions

- We have seen how to create anonymous functions using `lambda` and pass them as arguments.
- Now let's combine that with a function that itself returns another function (which we then bind to a variable).

```
? (defparameter foo '(11 22 33 44 55))

? (defun make-range-test (lower upper)
    #'(lambda (x)
        (and (> x lower)
             (< x upper))))


? (filter foo (make-range-test 10 30))

→ (11 22)
```

## Programming in INF4820

- In the IFI Linux environment, we have available Allegro Common Lisp, a commercial Lisp interpreter and compiler.
- We will provide a pre-configured, integrated setup with emacs and the SLIME Lisp interaction mode.
- Several open-source Lisp implementations exist, e.g. Clozure or SBCL; compatible with SLIME, so feel free to experiment (at some later point).
- First-time users, please spend some time studying basic keyboard commands, for example: `C-h t` and `M-x doctor RET`.
- See the getting started guide and emacs cheat sheet on the course page.
- Obligatory assignment 1 is out now, and due Wed. 9th Sept.
  - See course page or just run '`svn update`'.



## Next week

**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.