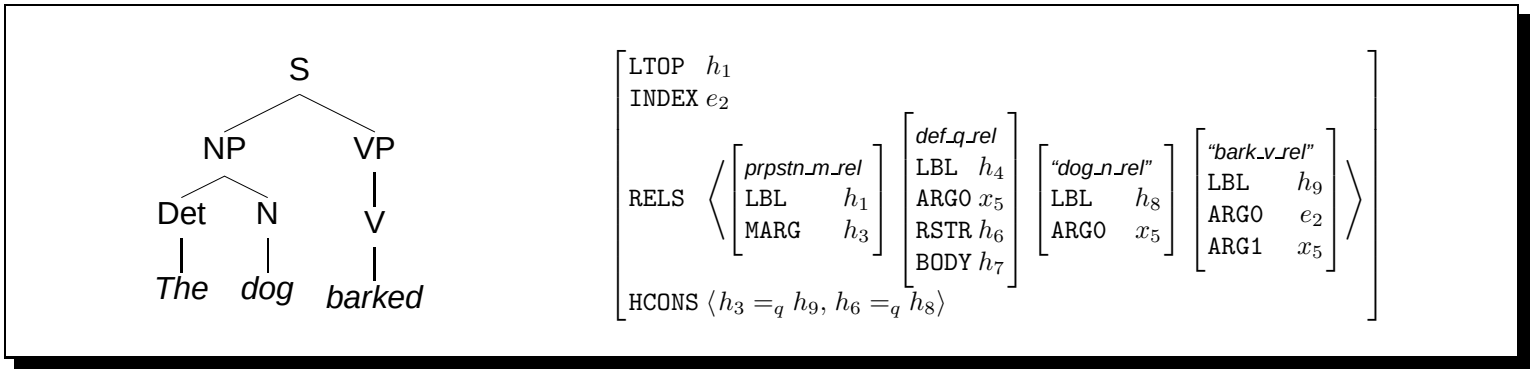


Computational Linguistics (INF2820 — Lisp)

(defun ? (n) (if (equal n 0) 1 (* n (! (- n 1)))))

Stephan Oepen

Universitetet i Oslo & CSLI Stanford

oe@ifi.uio.no

Why Common-Lisp for Implementation Exercises?

- Arguably most widely used language for 'symbolic' computation;
- easy to learn: extremely simple syntax; straightforward semantics;
- a rich language: multitude of built-in data types and operations;
- full standardization; Common-Lisp has been stable for a decade;
- LKB (experimentation environment) implemented in Common-Lisp;

→ *Ruby was a Lisp originally, in theory.* [Yukihiro Matsumoto; 2006].

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

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

Common-Lisp: Syntax

- Numbers: `42`, `3.1415`, `1/3`;
- strings: `"foo"`, `"42"`, `"(bar)"`;
- symbols: `pi`, `t`, `nil`, `foo`, `FoO`;
- lists: `(1 2 3 4 5)`, `()`, `nil`,

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

- Lisp manipulates *symbolic expressions* (known as 'sexps');
- sexps come in two fundamental flavours, atoms and lists;
- atoms include numbers, strings, symbols, structures, et al.;
- sexps are used to represent *both* data and program code;
- rather few 'magic' characters: '(', ')', '"', '\', ';', '#', '|', '`';
- all operators use *prefix* notation;
- symbol case does *not* matter.

Common-Lisp: Semantics (aka Evaluation)

- Constants (e.g. numbers and strings, `t` and `nil`) evaluate to themselves:

  ? `3.1415` → `3.1415` ? `"foo"` → `"foo"` ? `t` → `t` ? `nil` → `nil`

- symbols evaluate to their associated value (if any):

  ? `pi` → `3.141592653589793`

  ? `foo` → *error* (unless a value was assigned earlier)

- lists are function calls; the first element is interpreted as an operator and invoked with the *values* of all remaining elements as its arguments:

  ? `(* pi (+ 2 2))` → `12.566370614359172`;

- the `quote()` operator (abbreviated as ‘’) suppresses evaluation:

  ? `(quote (+ 2 2))` → `(+ 2 2)`

  ? `'foo` → `foo`

A Couple of List Operations

- `first()` and `rest()` destructure lists; `cons()` builds up new lists:

  ? `(first '(1 2 3))` → `1`

  ? `(rest '(1 2 3))` → `(2 3)`

  ? `(first (rest '(1 2 3)))` → `2`

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

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

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

- many additional list operations (derivable from the above primitives):

  ? `(list 1 2 3)` → `(1 2 3)`

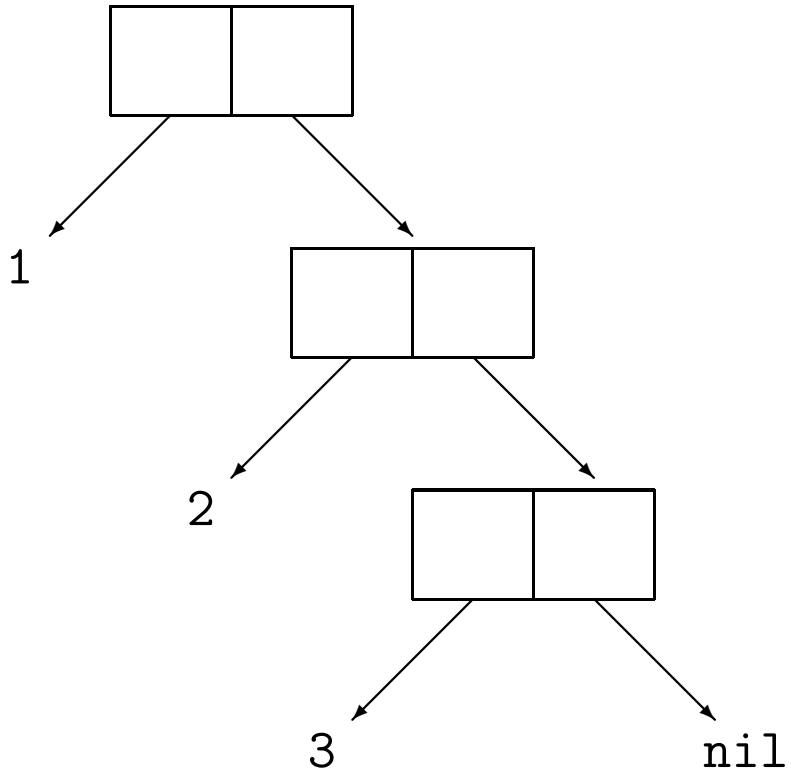
  ? `(append '(1 2 3) '(4 5 6))` → `(1 2 3 4 5 6)`

  ? `(length '(1 2 3))` → `3`

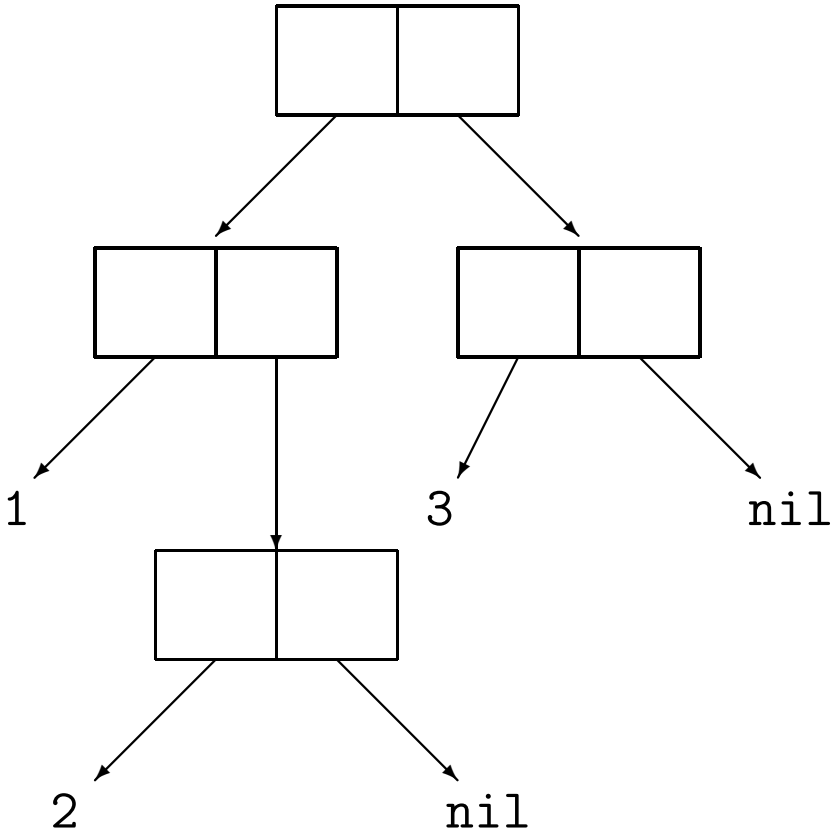
  ? `(reverse '(1 2 3))` → `(3 2 1)`

Lists: Internal Representation

(1 2 3)



((1 2) 3)




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

```
(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)` → `*FOO*`

  ? `*foo*` → `42`

- `setf()` associates ('assigns') a value to a symbol (a 'variable'):

  ? `(setf *foo* (+ *foo* 1))` → `43`

  ? `*foo*` → `43`

  ? `(setf *foo* '(1 1 3))` → `(1 1 3)`

- `setf()` can also alter the values associated to 'generalized variables':

  ? `(setf (first (rest *foo*)) 2)` → `2`

  ? `*foo*` → `(1 2 3)`

  ? `(setf (cons 0 *foo*) 2)` → *error*

Predicates — Conditional Evaluation

- A *predicate* tests some condition and evaluates to a boolean truth value; `nil` indicates *false* — anything non-`nil` (including `t`) indicates *true*:

  ? `(listp '(1 2 3))` → `t`

  ? `(null (rest '(1 2 3)))` → `nil`

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

  ? `(equal (cons 1 (cons 2 (cons 3 nil))) '(1 2 3))` → `t`

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

- conditional evaluation proceeds according to a boolean truth condition:

  ? `(if (numberp *foo*)`
  `(+ *foo* 42)`
  `(first (rest *foo*)))`
  → `2`

More Conditional Evaluation

- `if()` is fairly limited: exactly *one* sexp in its *then* and *else* branches:

  (`if` *test sexp sexp*)

- `when()` and `unless()` generalize *then* and *else* branches, respectively:

  (`when` *test sexp* ... *sexp*)

  (`unless` *test sexp* ... *sexp*)

- `cond()` allows an arbitrary number of conditions and associated sexps:

  (`cond`
  (*test*$_1$ *sexp* ... *sexp*)
  ⋮
  (*test*$_n$ *sexp* ... *sexp*)
  (`t` *sexp* ... *sexp*))

Defining New Functions

- `defun()` associates a function definition ('*body*') with a symbol:

  `(defun` *name* `(`$parameter_1$ ... $parameter_n$`)` *body*`)`

```
? (defun ! (n)
    (if (equal n 0)
      1
      (* n (! (- n 1)))))
  → !
? (! 0) → 1
? (! 5) → 120
```

- when a function is called, actual arguments (e.g. '`0`' and '`5`') are bound to the function parameter(s) (i.e. '`n`') for the scope of the function body;
- functions evaluate to the value of the *last* sexp in the function *body*.

Recursion as a Control Structure

- A function is said to be *recursive* when its *body* contains a call to itself:

```
(defun mlength (list)
  (if (null list)
    0
    (+ 1 (mlength (rest list)))))
```

- ? `(mlength '(a b))`

  0: `(MLENGTH (A B))`

  1: `(MLENGTH (B))`

  2: `(MLENGTH NIL)`

  2: returned `0`

  1: returned `1`

  0: returned `2`

  → 2

- *body* contains (at least) one recursive and one non-recursive branch.