Computational Linguistics II
— Grammars, Algorithms, Statistics —

Dan Flickinger
Oslo and Stanford Universities
danf@csli.stanford.edu

Tore Langholm
Universitetet i Oslo
torel@ifi.uio.no

Stephan Oepen
Oslo and Stanford Universities
oe@csli.stanford.edu

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;

→ for our purposes, (at least) as good a choice as any other language.

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

```
(defun ! (n)
  (if (= 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 (= 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.;
- lists 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):

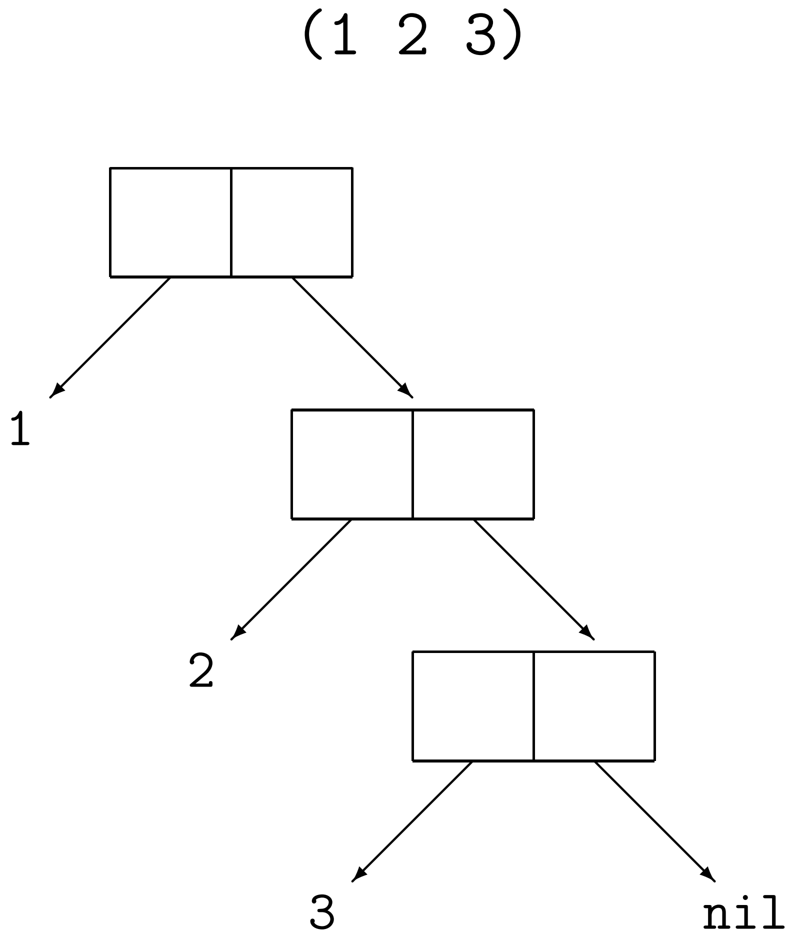
  ? `(list 1 2 3)` → `(1 2 3)`

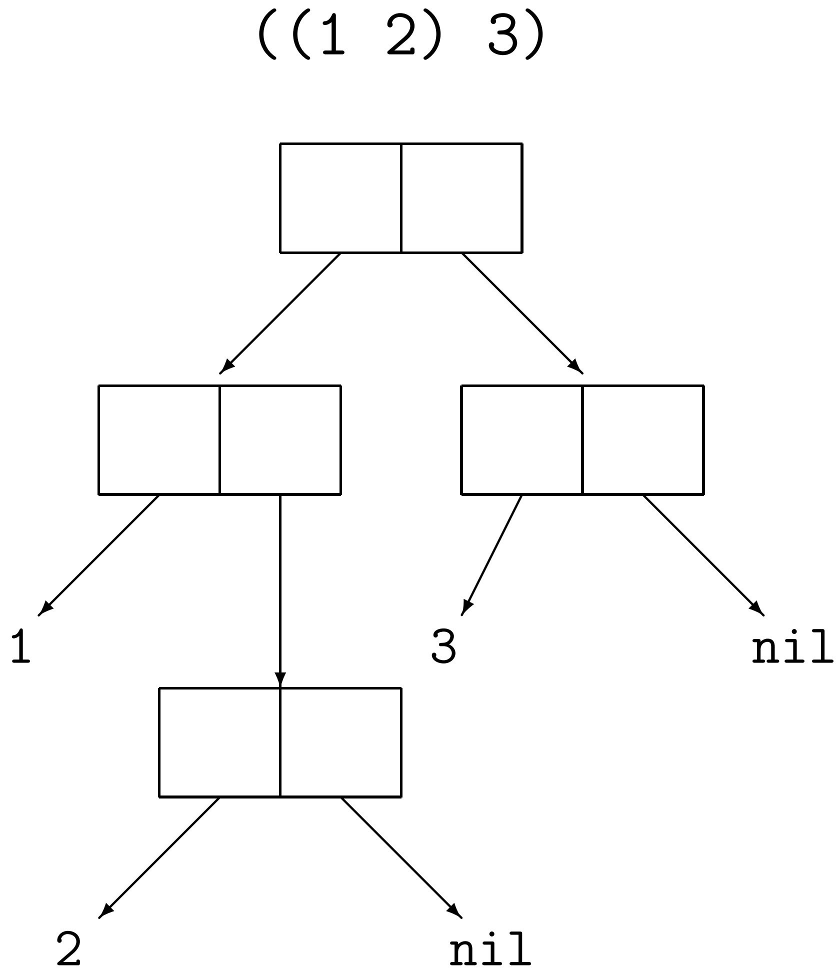
  ? `(append '(1 2 3) '(4 5 6))` → `(1 2 3 4 5 6)`

  ? `(length '(1 2 3))` → `3`

  ? `(reverse '(1 2 3))` → `(3 2 1)`

Lists: Internal Representation





(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 (= 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`
  $\longrightarrow$ 2

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

Iteration — Another Control Structure

- Recursion is very powerful, but at times *iteration* comes more natural:

```
(defun rules-deriving (category)
  (loop
      for rule in *grammar*
      when (equal (rule-lhs rule) category)
      collect rule))
```

Some `loop()` Directives

- `for` *symbol* { `in` | `on` } *list* — iterate *symbol* through *list* elements or tails;
- `for` *symbol* `from` *start* [`to` *end*] [`by` *step*] — count *symbol* through range;
- [{ `when` | `unless` } *test*] { `collect` | `append` } *sexp* — accumulate *sexp*;
- [`while` *test*] `do` *sexp*$^+$ — execute expression(s) *sexp*$^+$ in each iteration.

A Few More Examples

- `loop()` is extremely general; a single iteration construct fits all needs:

  ? `(loop for foo in '(1 2 3) collect foo)`
  → `(1 2 3)`

  ? `(loop for foo on '(1 2 3) collect foo)`
  → `((1 2 3) (2 3) (3))`

  ? `(loop for foo on '(1 2 3) append foo)`
  → `(1 2 3 2 3 3)`

  ? `(loop for i from 1 to 3 by 1 collect i)`
  → `(1 2 3)`

- `loop()` returns the final value of the accumulator (`collect` or `append`);
- `return()` terminates the iteration immediately and returns a value:

  ? `(loop for foo in '(1 2 3) when (evenp foo) do (return foo))`
  → 2

Abstract Data Types

- `defstruct()` creates a new *abstract data type*, encapsulating a structure:

```
? (defstruct rule
    lhs rhs)
→ RULE
```

- `defstruct()` defines a new *constructor*, *accessors*, and a type *predicate*:

```
? (setf *foo* (make-rule :lhs 'S :rhs '(NP PP)))
→ #S(RULE :LHS S :RHS (NP PP))
? (listp *foo*) → nil
? (rule-p *foo*) → t
? (setf (rule-rhs *foo*) '(NP VP)) → (NP VP)
? *foo* → #S(RULE :LHS S :RHS (NP VP))
```

- abstract data types *encapsulate* a group of related data (i.e. an ‘object’).

Vectors and Arrays

- Multidimensional 'grids' of data can be represented as *vectors* or *arrays*;
- `(make-array (` *rank*$_1$ ... *rank*$_n$`))` creates an array with *n* dimensions;

```
? (setf *foo* (make-array '(2 5) :initial-element 0))
→ #((0 0 0 0 0) (0 0 0 0 0))
? (setf (aref *foo* 1 2) 42) → 42
```

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	42	0	0

- all dimensions count from zero; `aref()` accesses one individual cell;
- one-dimensional arrays are called *vectors* (abstractly similar to lists).

Local Variables

- Sometimes intermediate results need to be accessed more than once;
- `let()` and `let*()` create temporary value bindings for symbols, e.g;

? `(defparameter foo 42)` → `FOO`

? `(let ((bar (+ foo 1))) bar)` → `43`

? `bar` → *error*

```
(let ((variable_1 sexp_1)
       ⋮
      (variable_n sexp_n))
  sexp ... sexp)
```

- bindings valid only in the body of `let()` (other bindings are *shadowed*);
- `let*()` binds *sequentially*, i.e. $variable_i$ will be accesible for $variable_{i+1}$.