

Algorithms for AI and NLP (INF4820 — Common Lisp)

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

Stephan Oepen and Jonathon Read

Universitetet i Oslo

{ oe | jread }@ifi.uio.no

Why Common-Lisp for (Symbolic) Programming?

- 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;
 - *Ruby was a Lisp originally, in theory.* [Yukihiro Matsumoto; 2006];
- 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 (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, Fo0;
- 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* program 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)

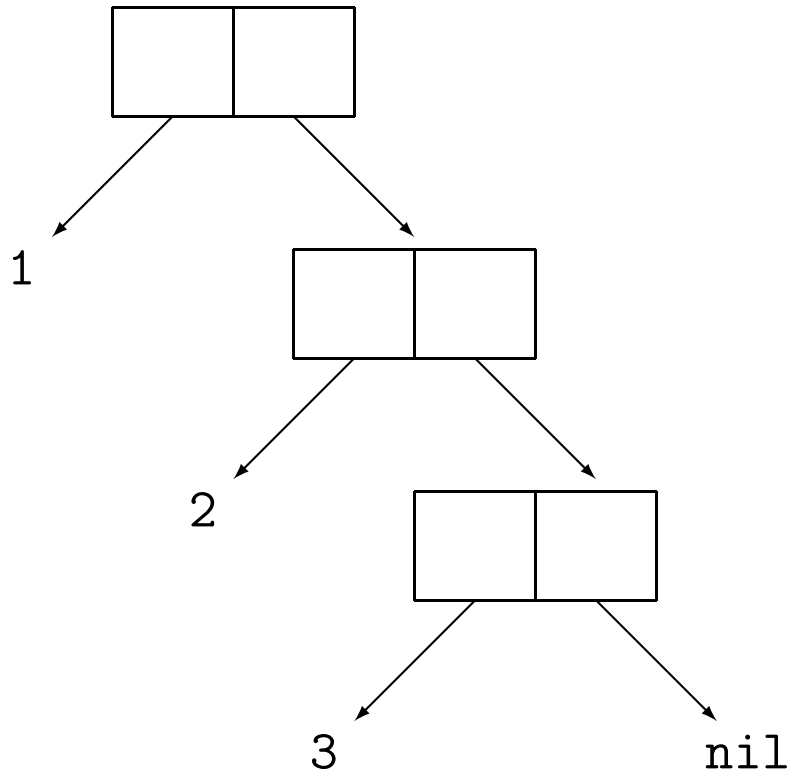
? `(length '(1 2 3))` → 3

? `(reverse '(1 2 3))` → (3 2 1)

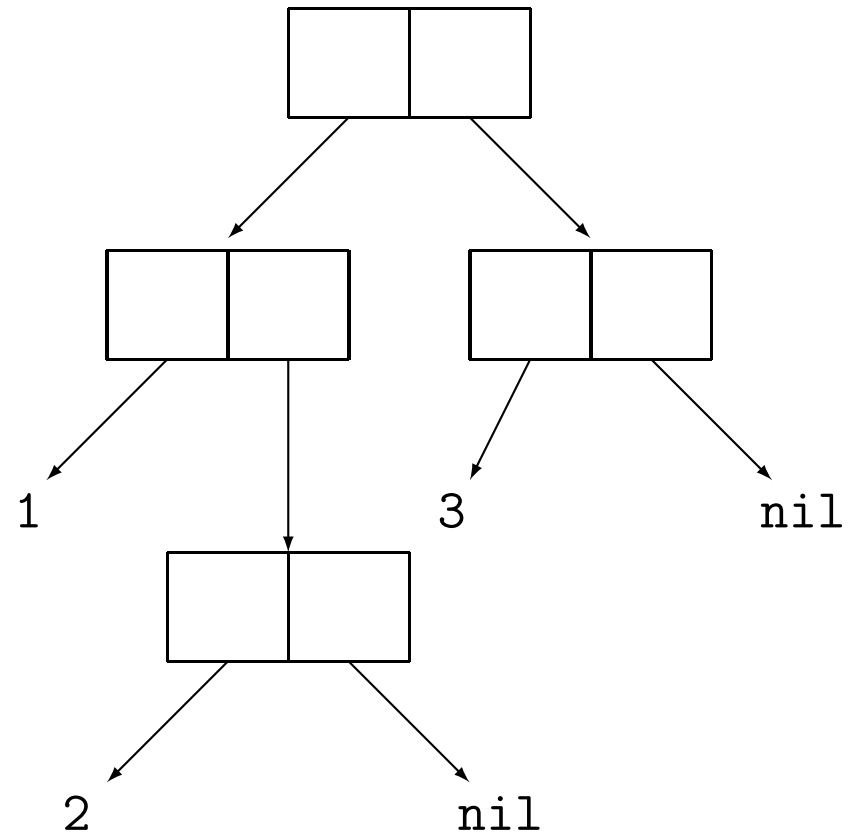


Lists: Internal Representation

(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) → *F00*
```

```
? *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
```



Defining New Functions

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

```
(defun name (parameter1 ... parametern) 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;
- a function evaluates 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.



Iteration — Another Control Structure

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

```
(defun odd-numbers (list)
  (loop
    for number in list
    when (oddp number)
    collect number))
```

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* in 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`



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`

`(test1 sexp ... sexp)`

⋮

`(testn sexp ... sexp)`

`(t sexp ... sexp))`



Vectors and Arrays

- Multidimensional 'grids' of data can be represented as *vectors* or *arrays*;
- `(make-array (rank1 ... rankn))` 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).



Some Objects are More Equal Than Others

- Plethora of equality tests: `eq()`, `eq1()`, `equal()`, `equalp()`, and more;
- `eq()` tests object identity; it is not useful for numbers or characters;
- `eq1()` is much like `eq()`, but well-defined on numbers and characters;
- `equal()` tests structural equivalence (recursively for lists and strings);

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

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

? (eq 42 42) → *dependent on implementation*

? (eq1 42 42) → t

? (eq1 42 42.0) → nil

? (equalp 42 42.0) → t

? (equal "foo" "foo") → t

? (equalp "F00" "foo") → t



Functions as First-Class Citizens

- Built-in functions (`member()`, `position()`, et al.) use `equal()` by default;
- but almost all take an optional `:test` argument to provide a predicate:

```
? (position "foo" '("foo" "bar")) → nil
```

```
? (position "foo" '("foo" "bar") :test #'equal) → 0
```
- `#'foo` is a short-hand for `(function foo)`, yielding a *function object*;
- function objects are first-class citizens, i.e. can be treated just like data;
- `funcall()` can be used to *invoke* a function object on a list of arguments:

```
? (funcall #' + 1 2 3) → 6
```
- many built-in functions also take a `:key` argument, an accessor function:

```
? (sort '((47 :bar) (11 :foo)) #'< :key #'first)
```

```
→ ((11 :F00) (47 :BAR))
```



Hash Tables: Yet Another Container Data Structure

- Lists are inefficient for indexing data, arrays restricted to numeric keys;
- *hash table* is a (one-dimensional) associative array; free-choice keys;
- any of the four (built-in) equality tests can be used for key comparison;

```
? (defparameter *map* (make-hash-table :test #'equal))  
→ *map*
```

```
? (gethash "foo" *map*) → nil
```

```
? (setf (gethash "foo" *map*) 4711) → 4711
```

```
? (gethash "foo" *map*) → 4711
```

- the more restricted the test, the more efficient will be hash table access;
- to iterate over hash table entries, use specialized `loop()` directives:

```
? (loop  
  for key being each hash-key in *map*  
  collect (gethash key *map*))
```



Input and Output — Side Effects

- Input and output, to files or the terminal, is mediated through *streams*;
- the symbol `t` can be used to refer to the default stream, the terminal:

```
? (format t "line: ~a; token '~a'.~%" 42 "foo")  
~> line: 42; token 'foo'.  
→ nil
```
- `(read stream nil)` reads one well-formed s-expression from *stream*;
- `(read-line stream nil)` reads one line of text, returning it as a string;
- the second argument to reader functions asks to return `nil` on end-of-file.

```
(with-open-file (stream "sample.txt" :direction :input)  
  (loop  
    for line = (read-line stream nil)  
    while line do (format t "~a~%" line)))
```



Abstract Data Types

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

```
? (defstruct cd
    artist title)
→ CD
```

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

```
? (setf *foo* (make-cd :artist "Dixie Chicks" :title "Home"))
→ #S(CD :ARTIST "Dixie Chicks" :TITLE "Home")
```

```
? (listp *foo*) → nil
```

```
? (cd-p *foo*) → t
```

```
? (setf (cd-title *foo*) "Fly") → "Fly"
```

```
? *foo* → #S(CD :ARTIST "Dixie Chicks" :TITLE "Fly")
```

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



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) → *F00*

? (let ((bar (+ *foo* 1))) bar) → 43

? bar → *error*

```
(let ((variable1 sexp1)  
      ⋮  
      (variablen sexpn))  
  sexp ... sexp)
```

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

