



# Computational Linguistics (INF2820 — More Lisp)

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

**Stephan Oepen**

Universitetet i Oslo & CSLI Stanford

oe@ifi.uio.no

# Defining New Functions

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

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

→ 2

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



# 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*<sub>*i*</sub> will be accessible for *variable*<sub>*i*+1</sub>.



# 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*<sup>+</sup> execute expression(s) *sexp*<sup>+</sup> 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
```



# 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)))
```



# Fine Points of Strings and Regular Expressions

- Need to *escape* double quote (") in strings, e.g. `"foo \"bar\" baz";`
- likewise for RE operators, to force literal match, e.g. `/\[a-z]\+\. /;`
- backslash is escape character for Lisp strings → `"\[a-z]\+\";`
- REs in Lisp represented as strings, thus need *two* levels of escaping.

- The Portable Perl-Compatible Regular Expressions package for Lisp;

? `(ppcre:all-matches-as-strings`

`"(\+|-)?[0-9,]+(\.[0-9]+)?"`

`"in 1994, the loss was at $4,711,4242, or -4.2% per share"`

→ `("1994," "4,711,4242," "-4.2")`

- many more functions in PPCRE library; see the on-line documentation.





# 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).



# 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').

