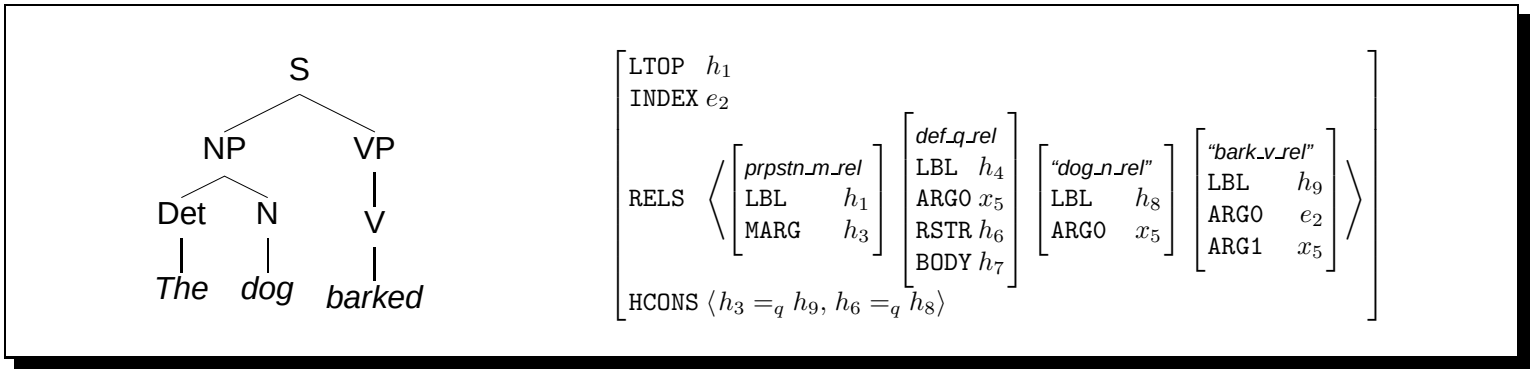

Computational Linguistics (INF2820 — More Lisp)

(defun ? (n) (if (equal 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* `(`*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.

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 2))` → `86`

  ? `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 $variable_{i+1}$.

Iteration — Another Control Structure

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

```
(loop
    for number in '(1 2 3 4 5 6 7 8 9)
    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
```

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 (not (null 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.