

# COMMON LISP Basics

Sven Brüßow



University of Potsdam  
Department of Linguistics

April 26, 2006



University of Heidelberg  
Department of Psychology

November 21, 2007

- ▶ Actually LISP refers to an entire family of **functional** programming languages.
- ▶ The name is taken from **List** processing. However, some funny fellows argue "Lots of **I**nsipid and **S**tupid **P**arentheses" is much more adequate.
- ▶ LISP is old. In fact, after FORTRAN it is the 2nd oldest high-level programming language.
- ▶ The original LISP was developed by John McCarthy in the late 1950s and first described in 1960 [5].

- ▶ COMMON LISP is one amongst a number of distinct LISP **dialects**.
- ▶ The term "common" refers to the LISP commonly agreed as standard.
- ▶ Hence, occasionally the term **ANSI** COMMON LISP occurs.
- ▶ ANSI = American National Standards Institute
- ▶ In general, when speaking about LISP tacitly COMMON LISP is meant.

- ▶ "To get started, imagine sitting in front of a computer terminal. When LISP is resting, doing nothing, it displays a *prompt* to tell you that it is waiting for you to type something." [6, 10]
- ▶ "A LISP interpreter is a machine that carries out processes described in the LISP language." [1, 2]
- ▶ Sometimes the notion **read-eval-print loop (REPL)** occurs. The term interpreter is deprecated, but we ignore the subtle differences.

## OpenMCL

```
Welcome to OpenMCL Version 1.0 (DarwinPPC32)!  
?
```

## CMUCL

```
CMU Common Lisp ... running on localhost  
With core: /usr/lib/cmucl/lisp.core  
Dumped on: Thu, 2005-11-17 14:45:02+01:00 on localhost  
...  
Type (help) for help or (quit) to exit.  
  
Loaded subsystems:  
  Python 1.1, target Intel x86  
  CLOS 18e (based on PCL September 16 92 PCL (f))  
*
```

- ▶ **Comments** can be set in order to let the listener ignore subsequent text.
- ▶ The **semicolon** `;` comments the rest of the line.
- ▶ Everything embraced by `#|` and `|#` comments entire blocks, not just lines.
- ▶ **Make use of comments!** Not in the listener directly, but in your external programs later on.

```
1 ? ; I am a comment
2 I'm not
3 > Error in process listener(1): Unbound variable: I
4 ...
5 1 > :pop
6
7 ? #|
8 I am a
9 multi-line
10 comment
11|#
12 end of multi-line comment
13 > Error in process listener(1): Unbound variable: END
14 ...
15 1 >
```

One of the most striking peculiarities apart of the numerous parentheses is the strict **prefix notation** of LISP.

### Example: Simple arithmetics

```
1 ? (+ 13 23)
2 36
3 ? (* 5 119)
4 595
5 ? (- 100 5 20)
6 75
7 ? (mod 20 7)
8 6
9 ? (mod (mod 20 7) 4)
10 2
11 ?
```

- ▶ The basic data types are called **(symbolic) expressions**.
- ▶ An expression is either an **(symbolic) atom**, or a **list**.
- ▶ An atom is either a **symbol**, or a **number** (integer, ratio, floating point).
- ▶ A list is either a **cons** or the **empty list** NIL.

## Some atoms:

**Strings** "I am an atom", "me too", "kein mann der einen bart hatte war jemals gluecklich \*stop\*", ...

**Numbers** 1, 2, ..., 3.14, ...

**Quoted symbols** 'foo 'bar, ...

**The empty list** NIL, '()

- ▶ **NOTE:** Lists as well as strings are of the **sequence** data type (yet another data type). Hence, many **primitive procedures** may be applied to both.
- ▶ **NOTE:** Non-empty list are not atoms! However, **character sequences** (viz. strings) are!

```
1 ? (length '(ein kein einen keinen))
2 4
3 ? (length "ein mann war durchaus gluecklich")
4 32
5 ? (reverse '(mann bart))
6 (BART MANN)
7 ? (reverse "mann bart")
8 "trab nnam"
9 ? (atom "mann bart")
10 T
11 ? (atom '(mann bart))
12 NIL
13 ? (elt "ein mann" 0) ; start with 0!
14 #\e
15 ? (elt '(ein mann) 0)
16 EIN
17 ?
```

- ▶ Both, data and procedures are considered as **symbolic expressions**.
- ▶ Everything embraced by a left and a right parenthesis is a **list**.
- ▶ For instance, `(+ 1 2 3)` is a list containing 4 elements **separated by spaces**.
- ▶ The first element in a list is considered as a procedure and the remaining elements as its arguments.
- ▶ If we want to stop LISP from taking the first element of some list as the name of some procedure to be applied to the subsequent elements, those lists have to be **quoted**.

► The primitive procedure SETF assigns values to symbols.

```
1 ? (setf num 100)
2 100
3 ? num
4 100
5 ? (setf wh-pron 'der)
6 DER
7 ? wh-pron
8 DER
9 ? (setf dp '(ein mann))
10 (EIN MANN)
11 ? dp
12 (EIN MANN)
13 ?
```

- ▶ SETF may take **symbol-value pairs** as arguments.
- ▶ Pairwise assignment is equivalent with sequential SETF calls for each symbol-value pair.

```
1 ? (setf npi "jemals"  
2     ppi "durchaus"  
3     npi-licensor '(kein keine)  
4     ppi-licensor '(ein eine))  
5 (EIN EINE)  
6 ? npi  
7 "jemals"  
8 ? npi-licensor  
9 (KEIN KEINE)  
10 ?
```

► **Make use of DEFVAR and DEFPARAMETER!**

```
? (defvar current)
```

```
CURRENT
```

```
? current
```

```
> Error in process listener(1): Unbound variable: CURRENT
```

```
...
```

```
1 > :pop
```

```
? (setf current 'keinen)
```

```
KEINEN
```

```
? current
```

```
KEINEN
```

```
? (defparameter *sentence*  
    "ein mann war jemals gluecklich")  
*SENTENCE*  
? *sentence*  
"ein mann war jemals gluecklich"  
? (defparameter *input*)  
> Error in process listener(1): Unbound variable: *INPUT*  
...  
1 >
```

**Lists can be constructed by means of the following primitives:**

- ▶ CONS
- ▶ APPEND
- ▶ LIST
- ▶ PUSH
- ▶ POP

**Only PUSH and POP alter symbol values directly!**

```
1 ? (setf slots '(cat polarity gapped))
2 (CAT POLARITY GAPPED)
3 ? (pop slots)
4 CAT
5 ? slots
6 (POLARITY GAPPED)
7 ? (cons 'cat slots)
8 (CAT POLARITY GAPPED)
9 ? slots
10 (POLARITY GAPPED)
11 ? (push 'cat slots)
12 (CAT POLARITY GAPPED)
13 ? slots
14 (CAT POLARITY GAPPED)
15 ?
```

## Lists can be accessed with the following primitives:

- ▶ REST, FIRST, SECOND, THIRD, ... (don't know how far)
- ▶ CAR, CDR (deprecated for FIRST and REST)
- ▶ LAST, NTHCDR

```
1 ? (setf words '(mann bart war hatte gluecklich))
2 (MANN BART WAR HATTE GLUECKLICH)
3 ? (first words)
4 MANN
5 ? (rest words)
6 (BART WAR HATTE GLUECKLICH)
7 ? (first (rest words))
8 BART
9 ?
```

- ▶ According to mathematical convention there is a subtle difference between **procedures** and **functions**
- ▶ "Anything a procedure has done that persists after it returns its value is called a **side effect**" [6, 22].
- ▶ Procedures without side effects are called **functions**.
- ▶ Hence, not all procedures are functions, though every function is a procedure.
- ▶ **We follow the common agreement and ignore the difference between procedures and functions!**

**For the sake of completeness ...**

**... some procedures**

- ▶ SETF, PUSH, POP, DEFUN

**... some functions** (still procedures, though!)

- ▶ CONS, APPEND, LIST, ATOM, MEMBER

- ▶ The side effect of DEFUN is to set up a **procedure definition** in the global environment.
- ▶ DEFUN takes as arguments a new **procedure name**, a list of **parameters** and a **body** consisting of **forms**.
- ▶ A form is an expression that you want to be **evaluated**.

```
1 ? (defun square (x) (* x x))
2 SQUARE
3 ? (square 4)
4 16
5 ? (defun increment (x) (setf x (+ 1 x)))
6 INCREMENT
7 ? (increment 1)
8 2
9 ?
```

- ▶ A **predicate** is a **procedure** that returns either T (true) or NIL (false).
- ▶ **Conditionals** in combination with predicates perform operations depending on the result of some **test**.
- ▶ Apart from those predicates defined on purpose, LISP provides a number of inbuilt predicates.
- ▶ Predicates LISP automatically provides are called **primitive predicates**.

## Some primitive predicates:

- ▶ NULL, ENDP
- ▶ ATOM
- ▶ SYMBOLP
- ▶ LISTP
- ▶ =, EQ, EQL, EQUAL
- ▶ MEMBER (not quite, returns a list)
- ▶ ODDP, EVENP

## Primitives for **conditional expressions**.

- ▶ IF
- ▶ COND
- ▶ WHEN
- ▶ UNLESS
- ▶ CASE

```
1 ? (setf lex-entry '(noun mann))  
2 (NOUN MANN)  
3 ? (when (eq (first lex-entry) 'noun)  
4       (second lex-entry))  
5 MANN  
6 ? (unless (not (eq (first lex-entry) 'noun))  
7       (second lex-entry))  
8 MANN  
9 ?
```

```
1 ? (setf noun 'bart)
2 BART
3 ? (if (equal noun 'mann) nil (setf noun 'mann))
4 MANN
5 ? noun
6 MANN
7 ? (if (equal noun 'mann) nil (setf noun 'mann))
8 NIL
9 ? (unless (equal noun 'mann) (setf noun 'mann))
10 NIL
11 ? noun
12 MANN
13 ? (setf noun 'bart)
14 BART
15 ? (unless (equal noun 'mann) (setf noun 'mann))
16 MANN
17 ? noun
18 MANN
19 ?
```

Suppose there was no primitive procedure `PUSH`:

```
1 (defun my-push (e l)
2   (if (listp l)
3       ; then
4       (setf l (cons e l))
5       ; else
6       'ERROR))
```

What is the problem with `MY-PUSH`?

```
1 ? (setf adv nil)
2 NIL
3 ? (my-push 'jemals adv)
4 (JEMALS)
5 ? adv
6 NIL
7 ? (push 'jemals adv)
8 (JEMALS)
9 ? adv
10 (JEMALS)
11 ?
```

- ▶ **Recursion** is a powerful technique that allows to define something in form of a smaller instance of itself.
- ▶ If a procedure calls itself this is referred to as a **recursive call**.
- ▶ Procedure descriptions containing recursive calls are said to be **recursive definitions**.

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

$$fak(n) = \begin{cases} 1 & \text{if } n = 0, n = 1 \\ n \times fak(n - 1) & \text{if } n > 1 \end{cases}$$

```
1 (defun fakultaet (n)
2   (if (equal n 0)
3       1
4       (* n (fakultaet (- n 1)))))
```

## What if there was no MEMBER primitive?

```
1 (defun my-member (elmt lst)
2   (cond ((endp lst) nil)
3         ((equal elmt (first lst)) t)
4         (t (my-member elmt (rest lst)))))
```

```
? (my-member 'a '(b c d e g))
```

```
NIL
```

```
? (my-member 'a '(b c d e a g))
```

```
T
```

```
?
```

- ▶ Beyond recursion, **mapping** is another important control strategy.
- ▶ Mapping primitives help to transform and filter elements in a list.
- ▶ For the moment we concentrate on MAPCAR.

```
(defparameter *conditions*  
  '(a "kein mann der einen bart hatte war jemals gluecklich")  
    (b "ein mann der keinen bart hatte war jemals gluecklich")  
    (c "ein mann der einen bart hatte war jemals gluecklich")  
    (d "kein mann der einen bart hatte war durchaus gluecklich")  
    (e "ein mann der keinen bart hatte war durchaus gluecklich")  
    (f "ein mann der einen bart hatte war durchaus gluecklich")))
```

```
(defun present-conditions (conditions)  
  (mapcar #'first conditions))
```

```
? (present-conditions *conditions*)  
(A B C D E F)  
?
```

```
1 (defparameter *lexicon*
2   '((affe noun) (auf prep)
3     (durchaus adv) (gluecklich adj)
4     (immer adv) (jemals adv)
5     (sein vainf) (war vafin)))
6
7 (defun cat (entry) (second entry))
8
9 (defun get-cat (lexicon)
10  (mapcar #'cat lexicon))
```

```
? (get-cat *lexicon*)
(NOUN PREP ADV ADJ ADV ADV VAINF VAFIN)
?
```

```
1 (defun filter-cat (cat lex)
2   (let ((result nil))
3     (mapcar #'(lambda (entry)
4               (when (equal cat (second entry))
5                 (setf result (cons (first entry)
6                                     result))))
7     lex)
8   result))
```

```
? (filter-cat 'adv *lexicon*)
(JEMALS IMMER DURCHAUS)
?
```

- ▶ **Iteration** is an **imperative** principle and **not a functional** one!
- ▶ Iteration can be replaced by **recursion**.
- ▶ *LISP* does not care and provides some wonderful primitives for iteration.

## What if there was no MEMBER primitive? – revisited

```
1 (defun my-member-i (elmt lst)
2   (dolist (e lst)
3     (when (equal e elmt)
4       (return T)))) ; Default return value is NIL!
```

```
? (my-member-i 'a '(b c d e f g))
```

```
NIL
```

```
? (my-member-i 'a '(b c d e a g))
```





```
T
```

```
?
```

```
1  
2 (defun present-conditions-x-times (times conditions)  
3   (let ((result nil))  
4     (dotimes (count times result)  
5       (dolist (condition conditions)  
6         (setf result (cons (first condition) result))))))
```

```
? (present-conditions-x-times 5 *conditions*)  
(F E D C B A F E D C B A F E D C B A F ... F E D C B A)  
? (present-conditions-x-times 1 *conditions*)  
(F E D C B A)
```

- ▶ Steele [3] provides a detailed reference of COMMON LISP.
- ▶ Winston and Horn [6] provide a more readable and most recommendable introduction (also covers CLOS).
- ▶ Another popular LISP dialect is SCHEME which is introduced in the great book *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman [1]
- ▶ Gazdar and Mellish [2] provide an introduction to Computational Linguistics utilizing LISP (cf. the PROLOG counterpart).
- ▶ Keene [4] provides the most common introduction to CLOS.

-  Harold Abelson and Gerald Jay Sussman with Julie Sussman.  
*Structure and Interpretation of Computer Programs.*  
MIT Press, Cambridge, Massachusetts, 1996.
-  Gerald Gazdar and Christopher Mellish.  
*Natural Language Processing in LISP: An Introduction to Computational Linguistics.*  
Addison-Wesley, Wokingham, England, 1989.
-  Jr. Guy L. Steele.  
*Common LISP: the language (2nd ed.).*  
Digital Press, 1990.
-  Sonya E. Keene.  
*Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.*  
Addison-Wesley, Reading, Massachusetts, 1989.



John McCarthy.

Recursive functions of symbolic expressions and their computation by machine, part 1.

*Communications of the ACM*, 3(4):184–195, 1960.



Patrick Henry Winston and Berthold Klaus Paul Horn.  
*LISP*.

Addison-Wesley, Reading, Massachusetts, 1993.