# Robot Programming with Lisp

## 2. Imperative Programming

Gayane Kazhoyan

Institute for Artificial Intelligence
Universität Bremen

21st October, 2014

# Outline

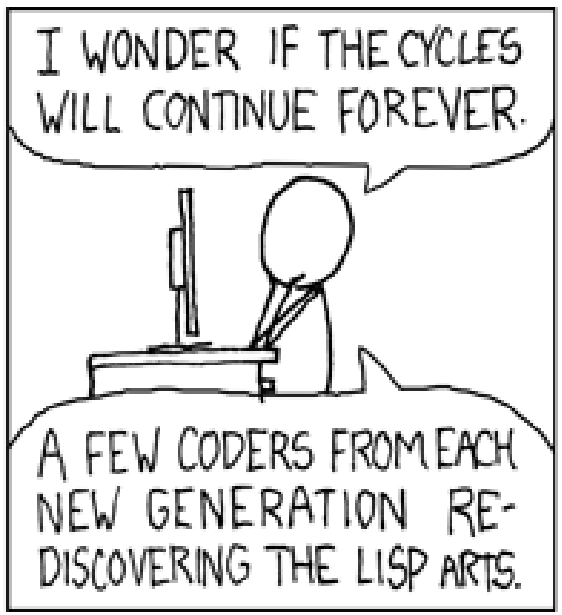
Theory

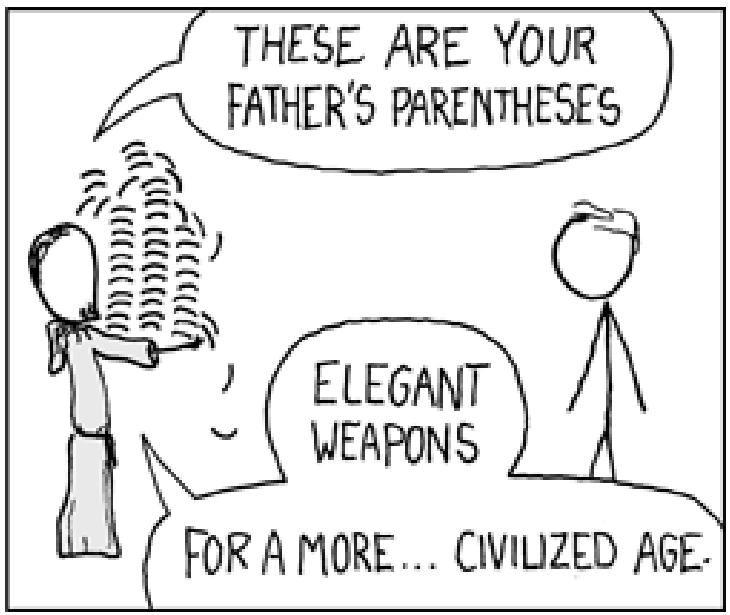
Assignment

# Lisp the Language

LISP ↔ LISt Processing language

Although, some say LISP ↔ Lots of Irritating Superfluous Parenthesis

# Technical Features

- *Dynamic typing*: specifying the type of a variable is optional
- *Garbage collection*: first language to have an automated GC
- *Functions as first-class citizens* (e.g. callbacks)
- *Anonymous functions*
- *Side-effects* are allowed, not purely functional as, e.g., Haskell
- *Run-time code generation*
- *Easily-expandable* through the powerful macros mechanism

# Short History

- 1958: Lisp as a theoretical language designed by John McCarthy (MIT)
- 1958: first Lisp interpreter implementation by Steve Russel (MIT)
- 1962: first Lisp compiler by Tim Hart and Mike Levin (MIT)
- End of 1960s: MacLisp (MIT), Interlisp (Xerox, Stanford, DARPA)
- 1975: Scheme (MIT)
- 1976: Emacs and EmacsLisp by Richard Stallman and Guy Steele
- 1970s - 2000s: Franz Lisp (UC Berkeley), NIL (MIT, Yale), AutoLISP (AutoCAD), Le Lisp (INRIA), PSL (Utah), CMUCL (CMU), T (Yale), EuLisp, Racket, SKILL, LFE (Lisp Flavoured Erlang), ISLISP (ISO standard), Arc, etc
- 1984: Common Lisp by Guy Steele (CMU)
- 2004: ANSI Common Lisp
- 2007: Clojure

# Hello World

## Java "Hello world":

```
public class HelloWorld {
  public static void main(String[ ] args) {
    System.out.println("Hello World!");
  }
}
```

## Lisp "Hello World"

```
"Hello World!"
```

# Polish Notation

Also known as **prefix notation**.

## Examples

```
(+ 1 2 3 4)

(sin 3.14)

(/ (+ 4 2) 3)

(list a b c)

(defun return-my-arg (arg)
  arg)

(return-my-arg 302)
```

# Numbers

## Integer

```
CL-USER> (describe 1)
1
  [fixnum]
```

## Float

```
CL-USER> (describe 1.0)
         (describe 1f0)
1.0
  [single-float]
CL-USER> 1f3
1000.0
```

## Double

```
CL-USER> (describe 1d0)
1.0d0
  [double-float]
```

# More Numbers

## Ratio

```
CL-USER> (describe (/ 1 3))
         (describe 1/3)
1/3
  [ratio]

CL-USER> (describe (/ 1.0 3))
0.33333334
  [single-float]
```

## Numeral Systems

```
CL-USER> #xFF
255

CL-USER> #b1111
15
```

# Chars and Strings

## Char

```
CL-USER> (describe #\Z)
#\Z
  [standard-char]
:_Char-code: 90
:_Char-name: LATIN_CAPITAL_LETTER_Z

CL-USER> (describe #\Ö)
#\LATIN_CAPITAL_LETTER_O_WITH_DIAERESIS
  [character]
:_Char-code: 214
:_Char-name: LATIN_CAPITAL_LETTER_O_WITH_DIAERESIS
```

## String

```
CL-USER> (describe "hello")
"hello"
  [simple-string]
```

# Variables and Symbols

## Variable

```
CL-USER> x
```

```
The variable X is unbound.
```

## Symbol

```
CL-USER> (describe 'x)
COMMON-LISP-USER::X
  [symbol]
```

## Keyword

```
CL-USER> (describe :x)
         (describe ':x)
:X
  [symbol]
X names a constant variable:
  Value: :X
```

# Booleans

## True and False

```
CL-USER> (describe NIL)
COMMON-LISP:NIL
  [null]
NIL names a primitive type-specifier

CL-USER> (describe T)
COMMON-LISP:T
  [symbol]
T names a primitive type-specifier
```

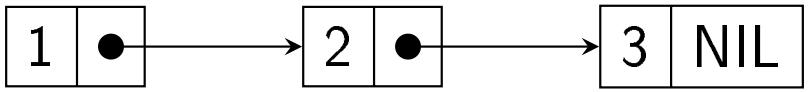
Everything, except `NIL`, is of type `T`.

# Lists

## List

```
CL-USER> (list 1 2 3)
(1 2 3)

CL-USER> *
(1 2 3)

CL-USER> (describe *)
(1 2 3)
  [list]

CL-USER> '(1 2 3)
(1 2 3)

CL-USER> '((1.1 1.2) (2.1 2.2) (some more stuff) (+ 1 3))
((1.1 1.2) (2.1 2.2) (SOME MORE STUFF) (+ 1 3))
```

# Lists Explained

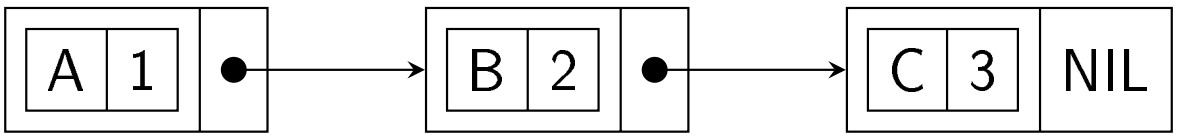



It's a linked list where each element has only 2 slots: `value` and `next-elem`. `next-elem` of the last element is `NIL`.
The slots are called *cons cells*.

## List and NIL

```
CL-USER> (describe '())
         (describe (list))
COMMON-LISP:NIL
  [null]

CL-USER> (type-of '(1 1 1))
CONS

CL-USER> (type-of '())
NULL
```

# Association Lists




It's a list where the first element of each cons cell is itself a cons cell.

## AList

```
CL-USER> '((A . 1) (B . 2) (C . 3))
((A . 1) (B . 2) (C . 3))

CL-USER>  (describe *)
((A . 1) (B . 2) (C . 3))
  [list]
```

# Arrays

## Vector

```
CL-USER> #(1 2 3)
#(1 2 3)
```

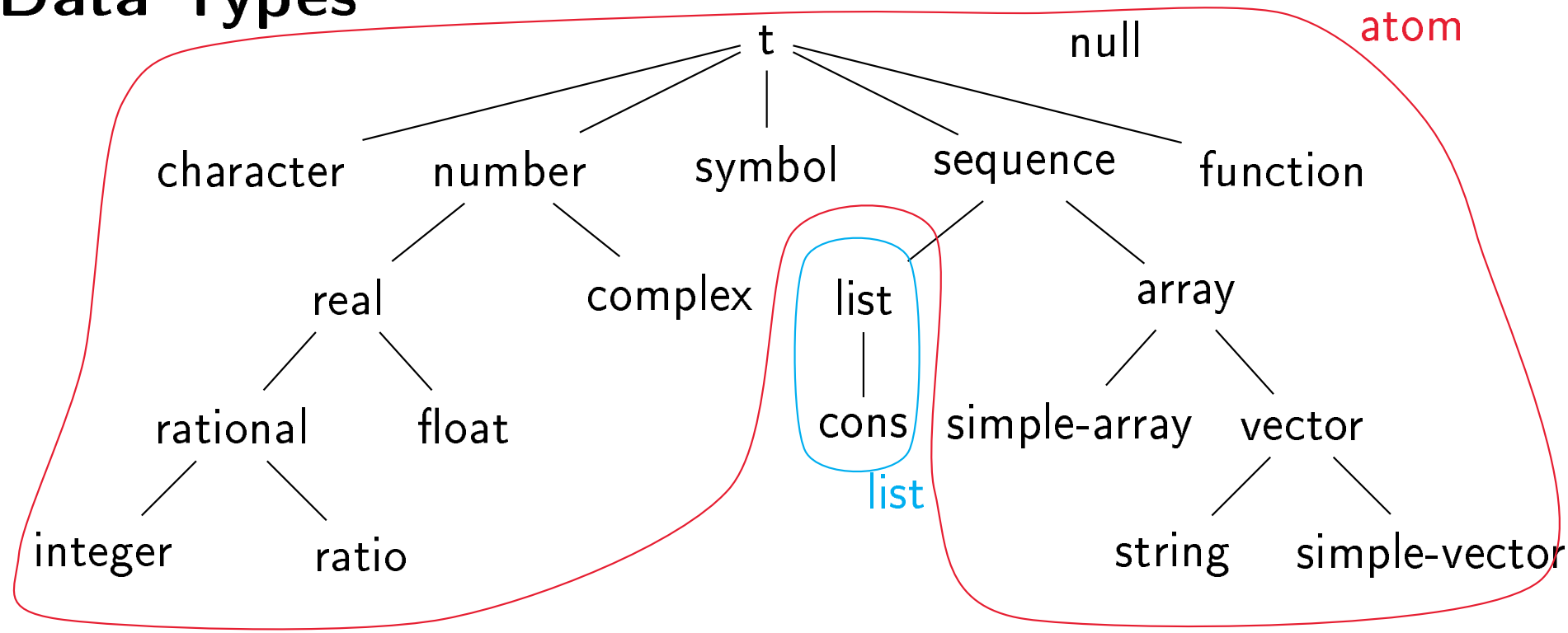
## Matrix

```
CL-USER> #2A((1 2) (3 4))
#2A((1 2) (3 4))
```

## Mutli-dimensional array

```
CL-USER> (make-array '(4 2 3)
            :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (3 1 2))
              ((g h i) (2 3 1))
              ((j k l) (0 0 0))))
#3A(((A B C) (1 2 3)) ((D E F) (3 1 2)) ((G H I) (2 3 1)) ((J K L)
(0 0 0)))
```

# Data Types




The diagram is very simplified. Also, the following is completely omitted:

- `standard-object` (CLOS)
- `stream` (files)
- `condition` (error handling), ...

# Data and code

Everything is either an `atom` or a `list`.

## Quoted constructs (both atoms and lists) are *data*:

```
CL-USER> '"abc"
"abc"

CL-USER> '(+ 1 2)
(+ 1 2)
```

## Everything else is *code*:

```
CL-USER> (+ 1 2)
3
```

## Conclusion:

`Run-time code generation and code manipulation done easily!`

# Code as Composition of Lists

Code is one big nested list. Depending on the first element, an S-expression is compiled into a function, special form or macro.

## Function

```
CL-USER> (print (list '+ 1 2))
(+ 1 2)


CL-USER> (eval (list '+ 1 2))
3
```

## Special Form

```
CL-USER> (list 'if t 1 2)
(IF T
    1
    2)
CL-USER> (eval *)
1
```

## Macro

```
CL-USER> (list 'defun 'return-a '(a) 'a)
(DEFUN RETURN-A (A) A)

CL-USER> (eval *)
RETURN-A
```

# More on Symbols

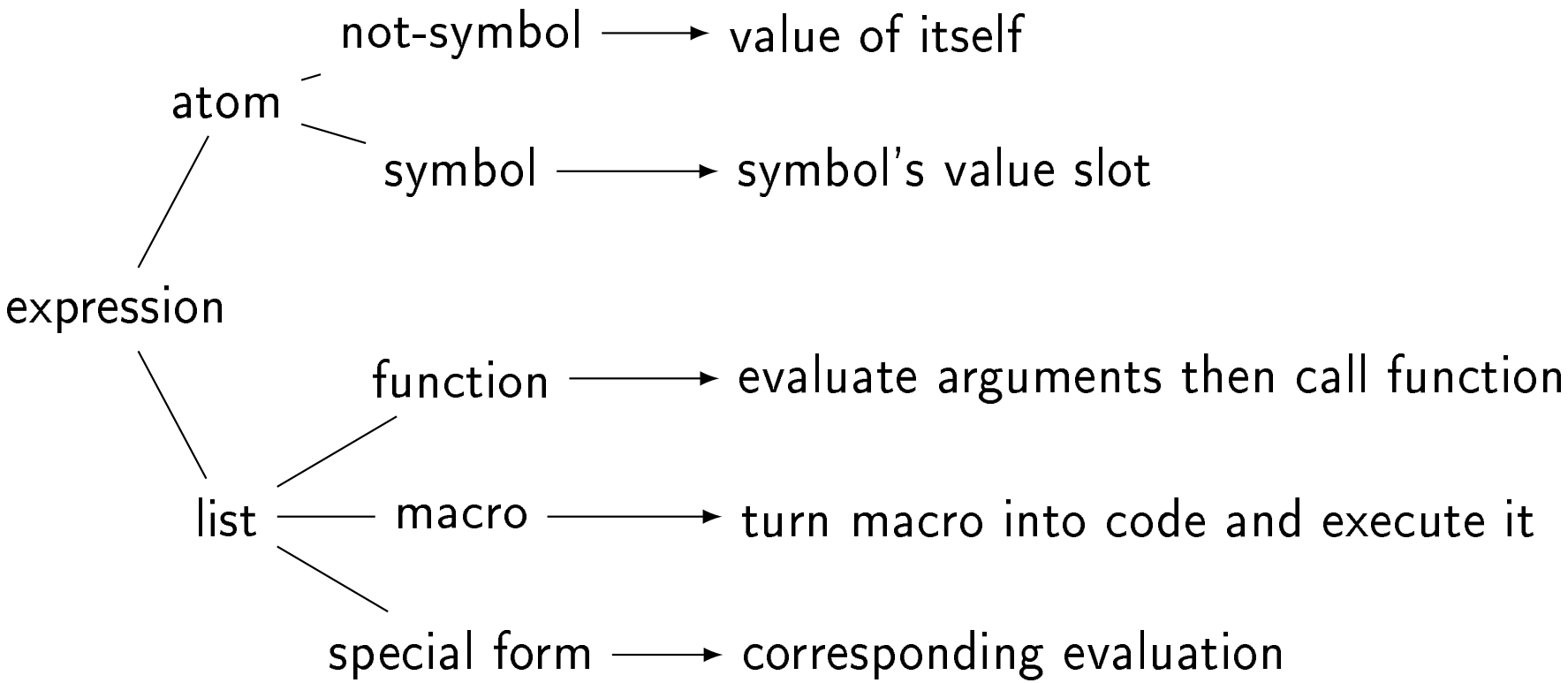
## Symbol

```
CL-USER> (setf my-symbol 1)
1
CL-USER> (defun my-symbol () 2)
MY-SYMBOL
CL-USER> (setf (get 'my-symbol 'my-property) 3)
3
CL-USER> 'my-symbol
MY-SYMBOL
```

| Symbol | |
|---|---|
| Field | Value |
| Name | my-symbol |
| Package | COMMON-LISP-USER aka CL-USER |
| Value | 1 |
| Function | #<FUNCTION MY-SYMBOL> |
| Property list | (MY-PROPERTY 3) |

To inspect an expression in the REPL:
*right click* → `inspect`.

# Read-Veal-Print Loop



# Imperative Programming

The *imperative programming* paradigm represents programs as sequences of commands.

One important property thereof is that the program has a *state* and the commands manipulate it to achieve a desired state.

Common Lisp has powerful means for imperative programming (e.g., very advanced looping mechanisms) but many traditionally imperative constructs are implemented differently in Lisp.

We'll consider both ways (*imperative vs. functional*) and then compare them.

# Special Variables

## Global Variables

```
CL-USER> (defvar *my-global-var* 'some-value "test variable")
*MY-GLOBAL-VAR*
CL-USER> *my-global-var*
SOME-VALUE
CL-USER> (setf *my-global-var* 23)
23
CL-USER> *my-global-var*
23
CL-USER> (incf *my-global-var*)
24
CL-USER> (defvar *my-global-var* 25)
*MY-GLOBAL-VAR*
CL-USER> *my-global-var*
24
```

Naming convention: `*the-variable-name*`.

# Special Variables [2]

## Parameters

```
CL-USER> (defparameter *my-param* 0.01)
*MY-PARAM*
CL-USER> *my-param*
0.01
CL-USER> (setf *my-param* 0.1)
0.1
CL-USER> *my-param*
0.1
CL-USER> (defparameter *another-param*)
error while parsing arguments to DEFMACRO DEFPARAMETER
CL-USER> (defparameter *my-param* 0.5)
*MY-PARAM*
CL-USER> *my-param*
0.5
```

# Special Variables [3]

## Constants

```
CL-USER> (defconstant +my-pi+ 3.14)
+MY-PI+
CL-USER> +my-pi+
3.14
CL-USER> (setf +my-pi+ 3.14159)
Execution of a form compiled with errors.
CL-USER> (defconstant +my-pi+ 3.14159)
+MY-PI+
The constant +MY-PI+ is being redefined
```

Naming convention: `+the-constant-name+`.

# Type Operations

## Predicates

```
(type-of 5) ⇒ INTEGER
(typep 5 'number) ⇒ T
(type-of #c(5 1)) ⇒ COMPLEX
(type-of 'nil) ⇒ NULL
(listp '(1 2 3)) ⇒ T
(symbolp 'a) ⇒ T
(type-of :k) ⇒ KEYWORD
(symbolp :k) ⇒ T
```

## Casts

```
(coerce '(a b c) 'vector) ⇒ #(A B C)
(coerce "a" 'character) ⇒ #\a
(coerce 7/2 'float) ⇒ 3.5
(coerce 7/2 'number) ⇒ 7/2
(coerce 7/2 't) ⇒ 7/2
(coerce 7/2 'null) ⇒ 7/2 can't be converted to type NULL.
```

# Comparing

## Casts

```
CL-USER> (> 2 1.5d0)
T
CL-USER> (<= 3.0d0 3)
T
CL-USER> (eq 1 1)
T
CL-USER> (eq 'bla 'bla)
T
CL-USER> (eq "bla" "bla")
NIL
CL-USER> (eq '(1 2 3) '(1 2 3))
NIL
CL-USER> (eql '(1 2 3) '(1 2 3))
NIL
CL-USER> (eql 1.0 1)
NIL
```

```
CL-USER> (equal '(1 2 3) '(1 2 3))
T
CL-USER> (equal "bla" "bla")
T
CL-USER> (equal "bla" "Bla")
NIL
CL-USER> (equalp "bla" "Bla")
T
CL-USER> (equal #(1 2 3) #(1 2 3))
NIL
CL-USER> (equalp #(1 2 3) #(1 2 3))
T
CL-USER> (= 2.4 2.4d0)
NIL
CL-USER> (string= "hello" "hello")
T
```

# Comparing [2]

| x | y | eq | eql | equal | equalp |
|---|---|---|---|---|---|
| 'a | 'a | T | T | T | T |
| 0 | 0 | ? | T | T | T |
| '(a) | '(a) | nil | nil | T | T |
| "ab" | "ab" | nil | nil | T | T |
| "Ab" | "aB" | nil | nil | nil | T |
| 0 | 0.0 | nil | nil | nil | T |
| 0 | 1 | nil | nil | nil | nil |

= is for comparing numbers of the same type.

`string=` is an advanced tool for comparing strings.

# List Operations

## Lists

```
CL-USER> (cons 1 (cons 2 (cons 3 nil)))
CL-USER> (list 1 2 3)
CL-USER> '(1 2 3)
(1 2 3)

CL-USER> (listp *)
T
CL-USER> (null **)
NIL
CL-USER> (null '())
T
CL-USER> (null '(()))
NIL
CL-USER> (member 2 '(1 2 3))
(2 3)
CL-USER> (member 2 '((1 2) (3 4)))
NIL
```

```
CL-USER> (defvar *my-list*
                 '(1 2 3 4 5))
*MY-LIST*
CL-USER> (first *my-list*)
1
CL-USER> (rest *my-list*)
(2 3 4 5)
CL-USER> (nth 4 *my-list*)
5
CL-USER> (fourth *my-list*)
4
CL-USER> (last *my-list*)
(5)

CL-USER> (push 0 *my-list*)
(0 1 2 3 4 5)
```

# AList Operations

## Association Lists

```
CL-USER> (cons (cons "Alice" "Jones")
               (cons (cons "Bill" "Smith")
                     (cons (cons "Cathy" "Smith")
                           nil)))
(("Alice" . "Jones") ("Bill" . "Smith") ("Cathy" . "Smith"))
CL-USER> (assoc "Alice" *)
NIL
CL-USER> (assoc "Alice" ** :test \#'string=)
("Alice" . "Jones")
CL-USER> (rassoc "Smith" *** :test \#'string=)
("Bill" . "Smith")
```

# Property Lists and Vectors

## Property Lists

```
CL-USER> (defvar *plist* '())
*PLIST*
CL-USER> (setf (getf *plist* 'key) 'value)
VALUE
CL-USER> *plist*
(KEY VALUE)
CL-USER> (setf (getf *plist* 'another-key)
               'another-value)
ANOTHER-VALUE
CL-USER> *plist*
(ANOTHER-KEY ANOTHER-VALUE KEY VALUE)
CL-USER> (setf (getf *plist* 'key)
               'new-value)
NEW-VALUE
CL-USER> *plist*
(ANOTHER-KEY ANOTHER-VALUE KEY NEW-VALUE)
```

## Vectors

```
CL-USER> #2A((1 2) (3 4))
#2A((1 2) (3 4))
CL-USER> (aref * 0 0)
1
CL-USER> (aref ** 1 1)
4
CL-USER> #(1 2 3 4 5 6)
#(1 2 3 4 5 6)
CL-USER> (setf (aref * 5) 9)
9
CL-USER> **
#(1 2 3 4 5 9)
```

## Format Statements, Streams and Files

```
CL-USER> (read)
hello world
HELLO
CL-USER> (read-line)
hello world
"hello world"
CL-USER> (format nil "symbol to ~a" 'string)
"symbol to STRING"
CL-USER> (format t "1 + 1 = ~a~%" (+ 1 1))
1 + 1 = 2
NIL
CL-USER>
(with-open-file (stream "~/.bashrc")
    (do ((line (read-line stream nil)
               (read-line stream nil)))
        ((null line))
      (print line)))
```

# Program Flow Constructs

## if, case, unless

```
CL-USER> (defvar *weather* 'rainy)
*WEATHER*
CL-USER> (if (eql *weather* 'rainy)
             (format t "I'm staying at home.")
             (format t "Let's go for a walk!"))
I'm staying at home.
NIL
CL-USER> (case *weather*
           (rainy "Stay home")
           (snowing "Go ski")
           (sunny "Got to the park")
           (otherwise "Hmmm..."))
"Stay home"
CL-USER> (setf *weather* 'very-nice)
VERY-NICE
CL-USER> (unless (eql *weather* 'rainy)
           (format t "Let's go for a walk!"))
Let's go for a walk!
```

# Program Flow Constructs [2]

## when, progn

```
CL-USER> (setf *weather* 'rainy)
RAINY
CL-USER> (if (eql *weather* 'rainy)
             (progn
               (format t "Let's go for a walk.~%")
               (format t "But don't forget your umbrella.~%")))
Let's go for a walk.
But don't forget your umbrella.
NIL
CL-USER> (when (eql *weather* 'rainy)
           (format t "Let's go for a walk.~%")
           (format t "But don't forget your umbrella.~%"))
Let's go for a walk.
But don't forget your umbrella.
NIL
```

# Program Flow Constructs [3]

## prog1, cond

```
CL-USER> (progn (setf *weather* 'who-cares)
                (format t "The weather today is ~a~%" *weather*))
The weather today is WHO-CARES
NIL
CL-USER> (prog1 (setf *weather* 'who-cares)
                (format t "The weather today is ~a~%" *weather*))
The weather today is WHO-CARES
WHO-CARES

CL-USER> (defvar *x* 1.5)
*X*
CL-USER> (cond ((< *x* 0) (exp *x*))
               ((< *x* 1) 1)
               (t *x*))
1.5
```

# Logical Operators

## and, or, not

```
CL-USER> (defparameter *threshold* 0.001)
*THRESHOLD*
CL-USER> (if (not (and (<= *threshold* 1) (> *threshold* 0)))
             (error "*threshold* should lie within (0; 1]~%"))
NIL
CL-USER> (if (or (> *threshold* 1) (<= *threshold* 0))
             (error "*threshold* should lie within (0; 1]~%"))
NIL
CL-USER> (unless (and (<= *threshold* 1) (> *threshold* 0))
           (error "*threshold* should lie within (0; 1]~%"))
NIL
```

# Looping

## dotimes, dolist, loop

```
CL-USER> (dotimes (i 10 (format t "the end~%"))
           (format t "~d " i))
0 1 2 3 4 5 6 7 8 9 the end
NIL
CL-USER> (defparameter *random* (loop repeat 10 collect (random 10000)))
*RANDOM*
CL-USER> (loop for i in *random*
               counting (evenp i) into evens
               counting (oddp i) into odds
               summing i into total
               maximizing i into max
               minimizing i into min
               finally (return (list min max total evens odds)))
(543 9756 48828 6 4)
CL-USER> (dolist (element *random* (format t "~%"))
           (format t "~a " element))
9756 7890 543 739 2371 8052 6586 1483 4202 7206
NIL
```

# Documentation

**Slime hotkeys** (in the REPL or `.lisp` file):

| | |
|---|---|
| `C-c C-d d` | describe symbol at point |
| `C-c C-d f` | describe function at point |
| `C-c C-d h` | open Hyperspec definition |
| `C-c C-d C-h` | list all Slime bindings for getting documentation |
| `M-.` | go into function definition |
| `M-,` | return one level up from the definition stack |
| `C-c M-m` | macroexpand expression at point |
| `C-c C-v i` | inspect presentation at point (only in REPL) (or *right click* → `inspect`) |

*Hint*: cursor is called "*point*" in Emacs.

# Outline

Theory

Assignment

# Assignment goals

Configuring Emacs


Lisp basics

# Task 1: Get info materials

- Download the compiler source code into a directory of your choice:

  `git clone https://github.com/sbcl/sbcl.git`

- Download ANSI Common Lisp documentation (the Hyperspec) and extract it into a directory of your choice:

  `ftp://ftp.lispworks.com/pub/software_tools/reference/HyperSpec-7-0.tar.gz`

# Task 2: Configure Emacs

- Remove (or backup) your Emacs configuration file if you have one.
  Usually it is `~/.emacs.d` or `~/.emacs` (~ ⇔ YOUR_HOME_DIRECTORY)
- Pull the updates from the original repository
  (to be executed in `lisp_course_material`):
  ```
  git clone https://github.com/sbcl/sbcl.git
  ```
- You will see a new directory that appeared there (`assignment_2`)
- Inside, there is an `.emacs.d` directory:
  ```
  assignment_2/misc/.emacs.d
  ```
- Copy it into your home or create a symbolic link:
  ```
  cd ~ && ln -s PATH_TO_.EMACS.D
  ```
- Open the file `.emacs.d/init.el` with an editor (Emacs, gedit)
- This is your Emacs configuration file. Go through it.
  Everything which starts with `FIXME` needs to be adjusted.
  (To search for "FIXME"-s in Emacs use `C-s FIXME`)
- Start Emacs. If there are errors, go back to `init.el`.

# Task 3: Lisp Homework

Your Lisp assignment is in `assignment_2/src`.

Follow the instructions therein.

Commit and push the changes once finished.

Good luck!

# Links

- Revenge of the nerds:
  `http://www.paulgraham.com/icad.html`
- Paredit cheat sheet:
  `http://pub.gajendra.net/src/paredit-refcard.pdf`
- Practical Common Lisp online book:
  `http://www.gigamonkeys.com/book/`
- The Little Schemer examples book:
  `http://www.ccs.neu.edu/home/matthias/BTLS/`

# Info summary

Assignment:

- Due: 26.10, Sunday, 23:59 German time
- Solution: will be available on the Git repo on Monday

Next class:

- Date: 28.10
- Time: 14:15
- Place: same room (TAB 2.63)
- Lecturer: Georg Bartels

# Q & A

Thanks for your attention!