Czech Technical University
Faculty of Informatics
Department of Theoretical Informatics

BI-PPA 2017 **Cvi od do Lispu** Date: 3. 11. 2017

What will we cover Basic constructs in Lisp (CLisp), function definitions, working with lists.

Lisp Basic Syntax Prefix, case **insensitive** insensitive language with lots of parentheses and applicative evaluation.
Basic building blocks are called **atoms**: atoms are numbers, characters (letters and other characters such as *, +, -, #) and strings.

symbols: `A,B,C,..., Z` ; characters
`0, 1, 2, ..., −1` ; integers
`1.0, 0.5, 3.14 ...` ; real numbers, must contain the decimal dot!
`1/2, 5/6, ...` ; fractions (will be converted to normal form)

strings: `"Hello", "+Hello'", ...`

arithmetics: +, −, `*`, `/`, `...`

logic: `not, or, and`

predicates: =, <, >, `equal, equalp, eq, eql, numberp, atom, zerop, listp, evenp, oddp`...

special symbols: `T` – true, `NIL` – false, ne, empty.
`quote, '` – takes the expression after ' without evaluating it, e.g. '(+ 2 3) returns (+ 2 3) and not 5.

conditions: `(if (`*condition*`)` *a b*`)`

functions: `(defun` *name* `(`*arguments*`)` *body*`)`

comments: `;` single line
`#|` multi-line, delimited with `|#` (or in some versions using `#||`, `||#`)

clisp: Basic interpreter commands:

start: `clisp`

exit: `(bye), (quit)` nebo `(exit)` ... all commands in clisp must be surrounded by parentheses.

evaluation: expression in parentheses, then enter.
e.g. `(+ 2 3)`, Enter, → `5`

loading file: `(load "file.lisp")` ... correct loading will be denoted by the following lines::
```
Loading file file.lisp
Loaded file file.lisp
T
```

help: `:h` to get the available commands

errors and typos: `:e` ... prints the text of last error

`:r3` or `:R3` ... return to interpreter's main loop.

Ex. 1. Exercise in writing expressions: Rewrite the following expressions to be valid in clipst (i.e. change to prefix, fix parenthes and let clisp evaluate them.

a) $(5 - 3 * (2 + 3))/(5 + 8/2 - 3)$ **Solution:** $(/(-5(*3(+23)))(-(+5(/82))3)) \rightarrow -5/3$

b) $(3 * 3 + 5 * 5 - 6/2)$ **Solution:** $(-(+(*33)(*55))(/62)) \rightarrow 31$

c) $(10/2 + 3/3 - 5)/(18 * (8/2 - 3))$ **Solution:** $(/(-(+(/102)(/33))5)(*18(-3(/82)))) \rightarrow -1/18$

Ex. 2. Write the definitions of the following simple functions (you are only allowed to use equalp, if, $+1$, -1, $<, >$, true, false, symbols and integers): **Solution:** solution can be found in separate file 1.lisp

a) addition of x and y,

b) subtraction of x and y,

c) multiplication of x and y,

d) division of x and y,

e) modulo of x and y,

f) square of x (returns $x * x$),

g) square root of x and y, (version II: as binary squaring)

h) maximum of x and y,

i) function which decides if first of two numbers is greater,

j) function which determines if x divides y,

k) factorial of n,

l) n-th fibonaci number, (version II: in linear time, version III: sub-linear time)

Solution: solution is in separate file 1-solution.lisp

Ex. 3. What will be the result of the following tuples and predicates `eq, eql, equal, equalp`. What is the difference between them?

a) 1 1 **Solution:** always returns T for all predicates

b) 1 1.0 **Solution:** always NIL, but for equalp

c) 1 4/4 **Solution:** always T

d) 1 (/ 4 4) **Solution:** always T

e) 5 (+ 2 3) **Solution:** always T

f) 'a 'A **Solution:** always T

g) 'ahoj 'Ahoj **Solution:** always T

h) 1.0 1.0 **Solution:** always T

i) '(2 3) '(2 3) **Solution:** eq, eql returns NIL, equal, equalp returns T

j) () () **Solution:** returns always T, for all predicates

Lisp basic structures - lists: Lists are basic data structure in Lisp. Their access is sequential, so specific access patterns are required.

Functions to work with lists: `cons, car, cdr, list, append, atom...`

pady seznam

```
(), nil          ; empty list
(1 2 3)          ; list of integerl 1, 2, 3
("foo" "bar")     ; list of strings
(x y z)          ; list of symbols x, y, z
(x 1 "foo")      ; list of symbol, integer and string
(+ (* 2 3) 4)    ; list of symbol, list (containing symbol, integer and integer) and integer.
```

Ex. 4. Wtite three different `cons` expressions that return (a b c). **Solution:** (cons a (cons b (cons c nil)))
(cons a (cons b (cons c ())))
(cons a (cons b '(c))))
(cons a '(b c))

Ex. 5. Draw the represetation of the following symbolic expressions in memory:

a) (A B (C D) E)

b) (A (B (C (D . E))))

c) (A (B . C) (D . E))

d) (A B (C . (D E)))

Ex. 6. Using `car` and `cdr` define function `fourth`, which returns fourth element in a list.
`(fourth '(1 3 8 6 5))` → `6` **Solution:** C-Lisp: (defun fourth (lst) (car (cdr (cdr (cdr lst)))))

Solution: Solution to all following exercises can be found in the file 1-solution.lisp

Ex. 7. Define function `contains`, which determines if given element is contained in a list.

- `(contains 6 '(1 2 3 4 5))` → `nil`
- `(contains 6 '(1 2 (8 3 6) 4 5))` → `nil`

Ex. 8. Define function `listLength`, which calculates length of a list.
`(length '(1 2 (3 4)))` → `3` **Solution:** C-Lisp:
(defun length (lst)
(if (eq lst ())
0
(+ 1 (length (cdr lst)))))

Ex. 9. Define function `middleItem`, which returns the middle item in a list (excluding sublists), and do so **without** using the length of a list.
`(middleItem '(1 2 3 4 5 6))` → `3` `(middleLitem '(1 2 (3 4) 5 6))` → `(3 4)`
Solution: C-Lisp:

(defun middleItem (lst)
(middleItem2 (cons () lst) lst)) ;we need to be one step behind, so put an empty list at the beginning of l so that we get correct result in case of l = nil
(defun middleItem2 (lst sez)
(if (equalp sez ())
(car lst)
(midleItem2 (cdr lst) (cdr (cdr sez)))))

Ex. 10. Define function `pushback`, which adds second list after the first one.
`(pushback '(1 2) '(5 6) )`→ `(1 2 5 6)` **Solution:** C-Lisp:
(defun pushback
(list-1 list-2)
(if (eq list-1 ()) list-2
(cons (car list-1) (pushback (cdr list-1) list-2))))

Ex. 11. Define function `max`, which returns the largest element in a list.
`max '( 6 8 1 3 9 2)` → `9` **Solution:** C-Lisp:
(defun max (list)
(if (equal list ())
()
(maxx (car list) (cdr list))))
(defun maxx (m list)
(if (eq list ())
m
(if (¡ m (car list))
(maxx (car list) (cdr list))
(maxx m (cdr list)))))

Ex. 12. Define function `atomy`, which calculates the number of atoms in an expression.

- `(atomy '(1 2 (3 4)))` → `4`
- `(atomy 1)` → `1`
- `(atomy '(1 2 (3 4 . 5) 6 . 7))` → `7`

Solution: C-Lisp: (defun atomy (arg)
(cond ((null arg) 0)
((atom arg) 1)
((eq 1 1) (+ (atomy (car arg)) (atomy (cdr arg)))))) ; else

Ex. 13. Define function `revlist`, which reverse the order of elements in list without changing the sublists. Solve also using tail recursion.
`(revlist '(1 2 3 (4 5)))` → `((4 5) 3 2 1)` **Solution:**

Ex. 14. Define function `genList`, which accepts integer x and number of repetitions n. The result is a list of n numbers, from x to $x + n - 1$.
`(genList 12 5)` → `(12 13 14 15 16)` **Solution:** C-Lisp:
(defun genList (x n)
(if (eq n 0)
'()

(cons x (genList (+ x 1) (- n 1)))))

Ex. 15. Define function `srlist`, which calculates a list of square roots of a list. Negative numbers will be ignored.
`(srlist '(11 -5 4 8 -7))` → `(3.3166247903554 2.0 2.8284271247461903)` **Solution:** (defun srList (lst)
(cond ((eq lst ()) '())
((¿= x 0) (cons (sqrt x) (srList (cdr lst))))
((¡ x 0) (srList (cdr lst)))))

Ex. 16. Define function `avgList`, which takes two lists, the result is list of their averages by elements. In case the length of the lists differ, extra values from the longer will be used unchanged.
`(avgList '(1 2 3 4) '(7 8 9 10))` → `(4 5 6 7)` **Solution:** C-Lisp:
(defun avgList (x y)
(cond ((and (null x) (null y)) '())
((null x) y)
((null y) x)
((eq 1 1) (cons (/ (+ (car x) (car y)) 2)
(avgList (cdr x) (cdr y))))))

Ex. 17. Define function `revAll`, that reverses all elements including sublists. Do not consider lists not finished with nil.
`(revAll '(1 (4 5 6) 3 ( 1 2 3)))` → `((3 2 1) 3 (6 5 4) 1)` **Solution:**