FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ
2. ÚVOD DO LISPU: ATOMY, SEZNAMY, FUNKCE,…

2011 Jan Janoušek

MI-FLP

L I S P - Introduction

L I S P

- What does LISP stand for ?
  - Lost In String of Parentheses ?
  - Looney Idiotic Stupid Professor?
  - LISt Processing !! (List is a fundamental data structure in LISP)

- Introduced by John McCarthy at MIT in 1959 as a model of computation and an implementation of recursive function theory.

- Based on lambda calculus + much more syntax properties and features added:
  - Named functions
  - Conditions
  - Functions can have more than one parameter
  - Data structures: lists, arrays, records,…
  - …
- Lisp characteristics:
  - Symbolic functional language, symbolic evaluations.
  - Programs can be manipulated as data - the input and the output of a program in Lisp can be other programs in Lisp.
  - Simple syntax.

- Lisp is the first programming language to have (most of these features have gradually been added to other languages):
  - Conditionals - if-then-else kind constructs.
  - A function type.
  - Recursion.
  - Garbage collection.
  - A symbol type.

Known applications written in Lisp

- Artificial intelligence (AI)
- Outside AI, well-known LISP applications:
  - Emacs Lisp - Emacs's extension language.
  - AutoCAD - computer-aided design system.
  - Parasolid - geometric modeling systém.
  - Yahoo! Merchant Solutions - e-commerce software.
  - Remote Agent software - deployed on NASA's Deep Space 1 (1998).
  - Script-Fu plugins for GIMP (GNU Image Manipulation Program).

Computers connected with Lisp

- HW implementations of basic Lisp functions:
  Symbolics, Inc. 1980s

- Connection Machine CM series of computers
  MIT, Thinking Machine, Inc.

- Lisp machine
  Lisp Machine, Inc.

Existing variations of LISP

- Lisp's uniform syntax makes it very easily extensible. Just write new functions and include them when launching Lisp.
- This led many groups to create their own Lisp dialects: BBN-Lisp, Franz Lisp, Interlisp-10, Le-Lisp, Lisp 1.5, Lisp/370, Lisp Machine Lisp, Maclisp, NIL, Scheme, ZetaLisp,…=> **INCOMPATIBILITY**.
- Purpose of **Common Lisp**: to unify the main dialects. Thus, it contains multiple constructs to do the same things. Currently, Common Lisp contains also features of imperative languages (variables, assign statement, iterative cycles,…).
- In our seminars we will use: Common-Lisp compilers/interpreters: **Allegro, LispWorks**

Programming in LISP

Objects in Lisp

- Every Lisp object is either an **atom** or a **list.**
- Programs and functions in Lisp are represented as lists.
- Examples of atoms:
  - numbers: 235.4 2e10 2/3
  - variables: foo 2nd-place *foo*
  - constants: pi t nil
  - strings, chars: "Hello!" #\a
  - arrays: #(1 "foo" A) #1A(1 "foo" A) #2A((A B C) (1 2 3))
  - structures: #s(person first-name jan last-name janousek)
  - bit vectors: #*10110
  - hash tables

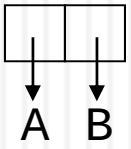
Lists

- Examples of **lists** (each list is written in parentheses):
  - ( ) NIL
  - (a)
  - (a john 34)
  - (lambda (arg) (* arg arg))
  - (1 (2 3) 4 (5 (6 7)))

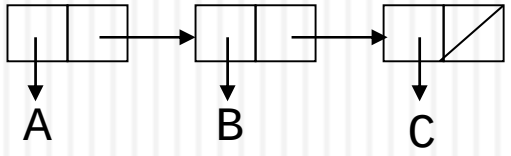
Lists and data representation

empty list: **`()`** = **`NIL`**

dotted-pair: **`(A . B)`**

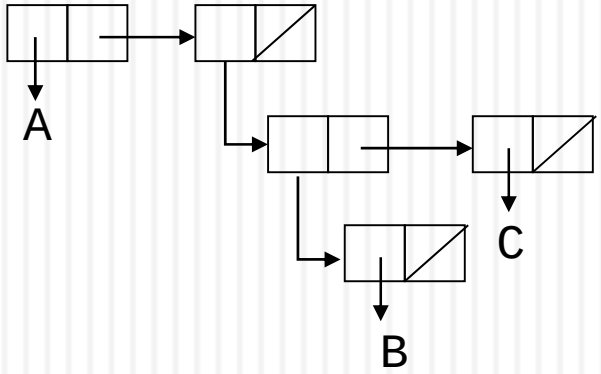


`(A B C)`



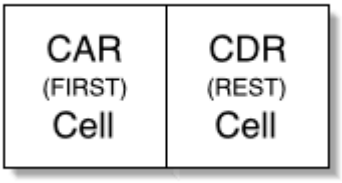
"cons-cell"

`(A ((B) C))`



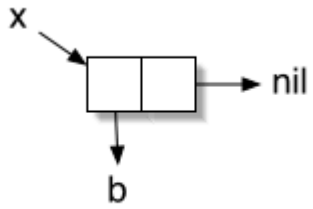
About Cons Cells

Lisp represents lists internally as singly linked lists. Each node in a Lisp list is called a **cons cell**. A cons cell consists of two parts: a **car** and a **cdr** (pronounced "could-er"). The car points to the element the node is holding. The cdr points to the next cons cell in the list, or else it points to **nil**, which represents the end of the list. Here's a picture of a cons cell:



The most important three functions which operate on cons cells are: **cons** which allocates a cons cell, **car** (otherwise known as **first**) which gives you the item the car is pointing to, and **cdr** (otherwise known as **rest**) which gives you the item the cdr is pointing to. Let's use them to create a simple list, namely `(b)`:

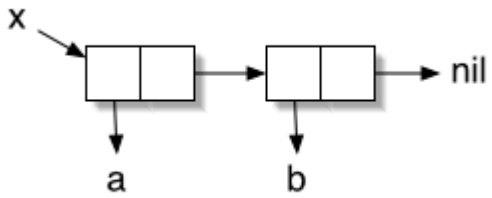
```
(setf x (cons 'b nil))     ;;; can also be done with (setf x '(b))
```



Now `x` is pointing to the first (and only) cons cell in a list. The only item in the list is `b`. The car of our cons cell is pointing to b. The cdr of the cons cell is pointing to nil. What do we get if we call `(car x)`? We get `b`. What do we get if we call `(cdr x)`? We get `nil`. By the way, `(car nil)` always returns `nil` and `(cdr nil)` always returns nil.

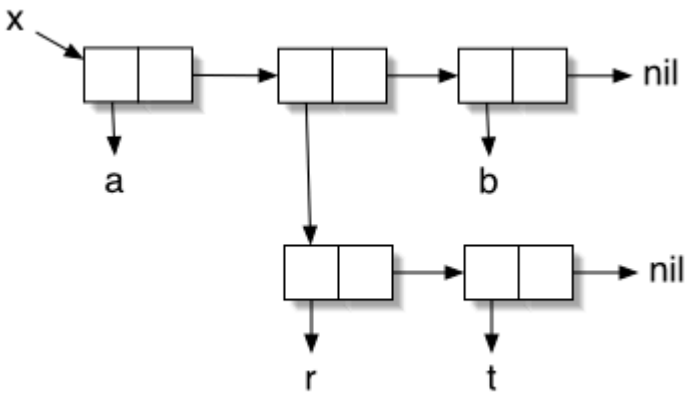
Here's a bigger list:

```
(setf x (cons 'a (cons 'b nil)))     ;;; or (setf x '(a b)) or (setf x (list 'a
'b))
```



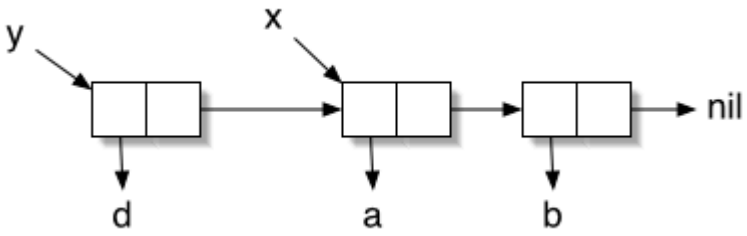
How do you do sublists? By pointing the car at the first cons cell in the sublist. For example:

```
(setf x '(a (r t) b))
```



Let's go back to `(setf x '(a b))`. What happens if we cons something onto that? As in:

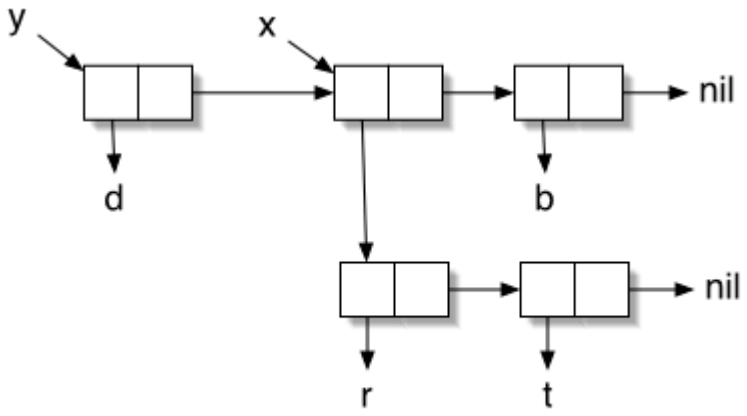
```
(setf x '(a b))
(setf y (cons d x))     ;;; y should now be '(d a b)
```



Notice that `x`'s original list is unaffected. We've just set up `y` to share it, along with an additional cons cell pointing to `d`.

You can modify cons cells using various destructive operators (most famously **rplaca** and **rplacd**). But the best way to do it is using **setf**. Remember that setf takes an expression and a value and "sees to it" that the expression returns the value. The way it does this for lists is by modifying the cons cells appropriately. For example, let's say we want to change it so that the first item in `x`'s list points to `'(r t)`, not `'a`. Assuming that `x` and `y` are set up as they were before,

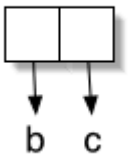
```
(setf (first x) '(r t))     ;;; or (setf (car x) '(r t))
```



Notice that `x` now reads `((r t) b)`. But `y` has also changed because it was sharing that list! Now `y` reads `(d (r t) b)`. Be careful with setf, it's a destructive operator with side effects that you have to think about first.

What happens if you have the cdr point to something other than another cons cell or nil? For example, what happens if you have the cdr point to, say, `c`? You could do this with cons, or by using setf to modify a list. For example:

```
(cons 'b 'c)     ;;; WHOA, c is not a list! Also: (setf (rest '(b)) c)
```

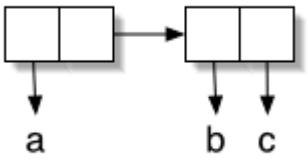


This is called a **dotted pair**. It's called this because of the way it prints out:

```
---> (b . c)
```

The item after the dot is what the last cdr in a list is pointing to, if not `nil`. What do you get if you call `(cdr (cons 'b 'c))`? You get the thing that the cdr is pointing to, namely, `c`. Same thing if you use rest instead of cdr of course. You can enter a dotted pair on the command line as well, as in: `'(b . c)` You can also have longer lists which end in a dotted pair:

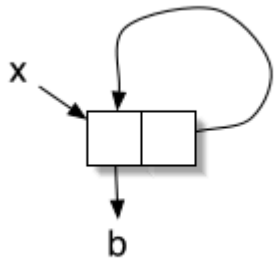
```
'(a b . c)
```



If you think about it, you'll realize that the only place where a dotted pair can appear is at the end of a list. So after the period there will be only one atom.

Lastly, you should be aware that you can use setf to do some funky stuff to cons cells. In class I showed how to use setf to manage queues. So some of this funky stuff is useful. Some of it is not. :-) Here's an example that's not all that useful:

```
(setf x '(b))
(setf (rest x) x)
```



That's right. You can make circular references. setf doesn't care. So be careful! How do you think Lisp prints this out for you? Try it out on osf1, but let's just say you'll want to be able to press control-C soon. :-)

Defining Lisp functions

(**defun** *name* (*formal_parameters*) *body*)

```
(defun square    (X)       (*     X      X))

(defun sum-of-squares (X Y)
  (+ (square X) (square Y)))

(defun hipotenusa (A B)
  (sqrt (sum-of-squares A B)))
```

Evaluating Lisp expressions in interpreters/compilers

Interpretation: expressions are evaluated in Listener (Allegro, LispWorks).

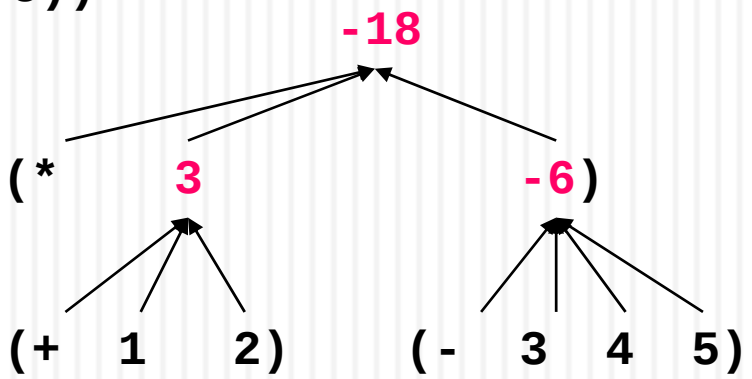
```
>(+ 2 5)    ; application of function + with two parameters, written as a list
7

>(defun square (X)(* X X))          ; function SQUARE is defined
SQUARE                               ; LISP IS CASE INSENSITIVE!

>(SQUARE 3)
9
```

Compilation: all expressions are evaluated during compilation.

Evaluating expressions

- **When evaluating an (arithmetic) expression:**
  - numbers evaluate to themselves
  - in a compound expression
    - **arguments (sub-expressions) are evaluated first**
    - **then the operation * function is applied**

```
(* (+ 1 2) (- 3 4 5))
```



More evaluation, Quote

Lisp (as a rule) evaluates all arguments before the main function is applied to them.

?How can we avoid that (if necessary for some reason)?

quote is the tool behind the apostrophe!

```
> (defvar X '(A B C))
X

> X
(A B C)

> (setf X (C B A))
Error: The variable B is unbound.

> (setf X (C 1 2))
Error: Undefined function C called with arguments (1 2).
```

same as **(quote (A B C))**

attempt to evaluate **B**

C considered a function

Syntactic sugar:

'exp ≡ (quote exp)

Conditions

$$abs(x) = \begin{cases} x & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -x & \text{if } x<0 \end{cases}$$

```
(defun abs (X)
  (cond ((> X 0) X)
        ((= X 0) 0)
        ((< X 0) (- X))
) )
```

```
(defun abs (X)
  (if (> X 0) X
    X
    (if (= X 0) 0 (- X))
) )
```

```
(cond   (p1  e1)
        (p2  e2)
          ...
        (pn  en)  )
```

```
(if predicate
        consequent
        alternative )
```

Predicates (return logical values T or NIL):

`=, /=, <, >, <=, >=, MINUSP, PLUSP, ZEROP, EVENP, ODDP`

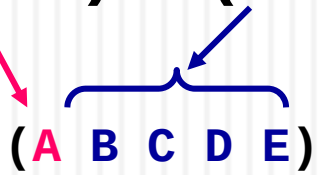
Processing lists

How to process lists and dotted pairs?

`(1 2 3 4)` `(A B C D E)` `(MI . MA)`

We have

selectors `(car s)` `(cdr s)` `(nth n s )` and more ...



constructors `(cons elm s)` `(list a b ... )` and more ...

predicates `(atom s)` `(eq a b )` or `eql`, `equal`

`(null s) = (eq s NIL )` ...

EXAMPLES:

```
defun swap1 (S)      ; interchange the first and second
  (cons (car (cdr S))                    ; cadr
        (cons (car S) (cdr (cdr S)))     ; cddr
) )

(defun swap2 (S)     ; interchange the first and second
  (cond ((null S) NIL)                 ; if S empty
        ((null (cdr S)) S)             ; if just one elm
        (T (cons (car (cdr S))         ; now safe
                 (cons (car S)
                       (cdr (cdr S))
) )      ) )     )

(defun swap3 (S)     ; interchange the first and second
  (cond ((null S) NIL)
        ((null (cdr S)) S)
        (T (cons (cadr S)
                 (cons (car S)
                       (cddr S))
) )      ) )     )
```

```
(defun select-atoms (S)    ; select all atoms of S
  (cond ((null S) NIL)
        ((atom (car S))
         (cons (car S) (select-atoms (cdr S))))
        (T (select-atoms (cdr S)))
) )

(defun test-positive (S)   ; test if all elms >0
  (cond ((null S) T)
        ((plusp (car S))
         (test-positive (cdr S)))
        (T NIL)            ; can be omitted
) )
```

```
(defun my-max1 (S)          ; calculate maximum elm
  (if (null S)
      'ErrorMax
      (my-max-iter (car S) (cdr S))
) )
(defun my-max-iter (TempMax S)
  (cond ((null S) TempMax)
        ((> (car S) TempMax)
         (my-max-iter (car S) (cdr S)))
        (T (my-max-iter TempMax (cdr S)))
) )

;; now find the difference
(defun my-max2 (S)          ; calculate maximum elm
  (cond ((null (cdr S)) (car S))
        ((> (car S) (my-max2 (cdr S))
         (car S))
        (T (my-max2 (cdr S)))
) )
```