FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ

3. LISP: ZÁKLADNÍ FUNKCE, SEZNAMY, POUŽÍVÁNÍ REKURZE, …

2011 Jan Janoušek
MI-FLP

OPP A
PRA HA PRA GUE PRA GA PRA G
EVROPSKÁ UNIE
Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

Comments in Lisp

- ; comments aligned to the same line
- ;; aligned for the next line and outside functions
- ;;; comments of headings of functions
- ;;;; comments of major sections, files

Condition expressions

=	equality of numbers
equal	structural isomorphism (same values)
eq	both arguments the same in memory
eql	identical objects – for numbers their values are compared; for structural objects their memory position are compared
and, or	logical and, or

More predicates

`(null s)`	T for an empty list
`(symbol s)`	T for atomic symbols
`(number s)`	T for numbers s
`(consp s)`	T for cons value s (i.e. it has car and cdr)
`(listp s)`	T for list s (maybe empty)

Selectors/accessors

- `cxx...xr` selectors - combinations of car/cdr
- `(caar x) = (car (car x))`   `(cadr x) = (car (cdr x))` ...
- Similarly `caaar, caadr, ..., cdddr, caaaar, caaadr, ..., cddddr`

`(length s)`	how many elements in s
`(nth n s)`	n-th element of list s (leftmost is 0th!)
`(nthcdr n s)`	n-th cdr
`(last s)`	last cons cell
`(butlast s)`	list s without the last element
`(butlast s n)`	n elements from the end removed

Standard, but could be defined:

```
(defun nth (N S)                    ; N-th element of list S
  (cond ((= N 0) (car S))                  ; car is 0th!
        (T (nth (1- N) (cdr S)))
) )

(defun nthcdr (N S)   ;n-th cdr
  (cond ((= N 0) S)
        (T (nthcdr (1- N) (cdr S)))
) )

(defun last (S)                            ; the last cons cell
  (cond ((null S) NIL)
        ((null (cdr S)) S)
        (T (last (cdr S)))
) )

(defun butlast (S)    ; list S without the last element
  (cond ((or (null S) (null (cdr S))) NIL)
        (T (cons (car S)
                 (butlast (cdr S))
) )      )  )
```

More functions

- `(member` *Elm Lst*`)` — looks for Elm in Lst, returns the part starting with Elm
- `(remove` *Elm Lst*`)` — deletes Elm from Lst (returns a copy)
- `(subst` *New Old Lst*`)` — substitutes New for Old in Lst (returns a copy)
- `(nsubst` *New Old Lst*`)` — substitutes New for Old in Lst (returns a modified Lst)
- `(reverse` *Lst*`)` — reverses Lst (returns a fresh list)

Again, they could be defined:

```
(defun member  (Elm Lst)
  (cond ((null Lst) NIL)
        ((eql Elm (car Lst)) Lst) ; eq, eql, equal ...
        (T (member Elm (cdr Lst)))
) )

(defun remove  (Elm Lst)
  (cond ((null Lst) NIL)
        ((eql Elm (car Lst))    ; Elm found?
         (remove Elm (cdr Lst)))        ; remove from rest
        (T (cons (car Lst)              ; else copy car
                 (remove Elm (cdr Lst)))
) )

(defun subst  (New Old Lst)
  (cond ((null Lst) NIL)
        ((eql Old (car Lst))
         (cons New
               (subst New Old (cdr Lst))))
        (T (cons (car Lst)
                 (subst New Old (cdr Lst))))
) )     ; what about substitution in all levels ???
```

Again, they could be defined:

```
(defun nsubst (New Old Lst) ; a modifying substitution
  (cond ((null Lst) NIL)
        ((eql Old (car Lst))
         (setf (car Lst) New)
         (setf (cdr Lst) (nsubst New Old (cdr Lst)))
        Lst)
        (T (setf (cdr Lst) (nsubst New Old (cdr Lst)))
           Lst)
) )        ; calls for some improvement ...

;; How do we reverse a list ??? Recursion classics …
(defun reverse  (Lst)
  (if (null Lst) NIL
    (append (reverse (cdr Lst))
            (list (car Lst)))))

(defun append (X Y)
  (if (null X) Y
    (cons (car X)
          (append (cdr X) Y))))
```

Again, they could be defined:

```
;; Can we do reverse any better ???
(defun reverse  (Lst)
  (rev-iter Lst Nil))

(defun rev-iter (Lst Acc)  ;tail-recursive version
  (if (null Lst) Acc
      (rev-iter (cdr Lst) (cons (car Lst)
Acc))))
```

Computation:

```
CL-USER 1 > (rev S)
0 REV > ...
  >> LST : (A B C)
  1 REV-ITER > ...
    >> LST : (A B C)   >> ACC : NIL
    2 REV-ITER > ...
      >> LST : (B C)   >> ACC : (A)
      3 REV-ITER > ...
        >> LST : (C)   >> ACC : (B A)
        4 REV-ITER > ...
          >> LST : NIL   >> ACC : (C B A)
        4 REV-ITER < ...
          << VALUE-0 : (C B A)
      3 REV-ITER < ...
        << VALUE-0 : (C B A)
    2 REV-ITER < ...
      << VALUE-0 : (C B A)
  1 REV-ITER < ...
    << VALUE-0 : (C B A)
0 REV < ...
  << VALUE-0 : (C B A)
```

In a simplified presentation:

```
(rev (A B C))
 (rev-iter  (A B C)  () )
  (rev-iter  (B C)  (A)  )
   (rev-iter  (C)  (B A)  )
    (rev-iter ()  (C B A)  )    →   (C B A)
    returned (C B A)
   returned (C B A)
  returned (C B A)
 returned (C B A)
returned (C B A)
```

Recursion

General rule of recursive design

1. first test for **trivial cases** (such as 0, 1, NIL, atom, …)
2. then develop branches with recursive calls in which the procedure is applied to reduced arguments (such as `(1- N), (cdr S),…`)

Important note: recursion is implemented with the use of pushdown store (stack) data structure.

A simple example

```
;;; copying a list (the highest level only)
(defun Copy (S)
 (if (null S) NIL
   (cons (car S) (Copy (cdr S))) ))


;;; Now copying at all levels
(defun CopyAll (S)
  (cond ((null S) nil)
        ((atom S) S)   ; remember this case!!
        ((cons (CopyAll (car S))
               (CopyAll (cdr S)))) ))
```

Taxonomy of recursion

- **nested recursion**

```
(defun Ack (m n)  ;(Ack 4 5) stack overflow (exponential)
   (cond      ((zerop m) (1+ n))
              ((zerop n) (Ack (1- m) 1))
              (T (Ack (1- m) (Ack m (1- n)))) ))
```

- **tree recursion**

```
(defun Fib (n)              ; (Fib 35) took 26.328 s
   (if (< n 2) n
       (+ (Fib (- n 1)) (Fib (- n 2)))
) )
```

Taxonomy of recursion

- **linear recursion**

```
;; just one recursive call
(defun linLen (s)
 (if (null s) 0
     (1+ (linLen (cdr s))) ; deferred operation
) )
```

- **tail recursion**

```
(defun trLen (s) (tailLen s 0))
;; one recursive call, no pending operations
(defun tailLen (s acc)
  (if (null s) acc
      (tailLen (cdr s) (1+ acc))
) )

;; acc - accumulator
```

Tail recursion

- nothing to do after the function returns except returning its value

- the most efficient type of recursion from the implementation point of view:
  - Saving a new stack frame for each recursive call is a waste! The current stack frame can be reused. (All better compilers of functional languages always do tail-recursion optimisation.)

- can be mechanically transformed into explicit iteration

Tail recursion

- Another example:

```
(defun factorial (N)
;;;"Compute the factorial of N."
(if (= N 0)
1
(* N (factorial (- N 1)))))

(defun fast-factorial (N)
;;;"A tail-recursive version of factorial."
(fast-factorial-aux N 1))

(defun fast-factorial-aux (N ACC)
;;;"Multiply A by the factorial of N."
(if (= N 0) A (fast-factorial-aux (- N 1) (* N A))))
```

Examples

to be discussed using the board…