Lab 1

Pure Lisp - recursion

1.0 INTRODUCTION

This lab material gives the introduction to Lisp with an orientation to its use for AI-labs. We assume all students to have programmed in an imperative language (C, C++, Ada, Java etc.) and a first course in data structures and algorithms.

We will emphasis on the functional programming paradigm and work a lot with recursion.

At IDA we use Franz Allegro Common Lisp system, which is integrated in the emacs-editor. There is a strong recommendation to bring a Lisp system to your home computer. See the www-pages for the course.

There is a lot of exercises in this material, but only ASSIGNMENTS should be given to the lab assistant for evaluation and examination.

1.1 TO START

First you must load a module to get access to Allegro CL. See the course’s www-page under LISP and Resources. There you can find how to install the Lisp system and how to start and use it.

On the www-pages for the course you will find links to the Common Lisp Standard, where you can find information about Lisp functions.

1.2 FIRST START

What is the result of the following expressions. Solve them first on paper and then verify your result.

12
-12
1/3 ; Ratio
(+ 12 45)
(+ 12 45 78) ; Arbitrary number of arguments
(+ 12 (- 45)) ; Unary minus
(+ 12 (- 45 78)) ; Binary minus
(1+ (* 2 3))
(1- (+ 2 3))
(/ 35 13)
(/ 35.0 13.0) ; Integer
(/ 34 5) ; Ratio
(float (/ 34 5)) ; Floating point number
(- (/ 1 (/ (+ 2 3) (* 4 5))) 6)
1.3 INFIX TO PREFIX NOTATION

Test corresponding Lisp expressions:

<table>
<thead>
<tr>
<th>INFIX FORM</th>
<th>LISP-FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+2·3</td>
<td>(1+2)·3</td>
</tr>
<tr>
<td>(1+2)·3</td>
<td>2·4+(6-8)</td>
</tr>
<tr>
<td>2·4+(6-8)</td>
<td>5+(−3+4)</td>
</tr>
<tr>
<td></td>
<td>6+2/5</td>
</tr>
</tbody>
</table>

1.4 DEFINITION OF FUNCTIONS

Exercise 1.4a. Define a function `average` with two arguments which returns the average value.

```lisp
(average 3 7)  => 5
(average 1 2)  => 3/2
```

Exercise 1.4b. Define a function `my-max` with two arguments which returns the largest value.

```lisp
(my-max 3 7)  => 7
```
1.5 RECURSION

There are two models for recursive functions: a recursive and an iterative process.

Exercise 1.5a. Define a function \(\text{pow}\) which calculates \(x^y\). Do two solutions, one which describes a recursive process and one as an iterative process. Use trace to look at the results.

\[
\begin{align*}
\text{(pow 2 3)} &= 2^3 = 2 \times 2 \times 2 \\
&\Rightarrow 8 \\
\text{(pow 5 2)} &= 5^2 = 5 \times 5 \\
&\Rightarrow 25
\end{align*}
\]

Exercise 1.5b. Define a function \(\text{add-to}\) which adds the first \(n\) natural numbers.

\[
\text{(add-to n)} = 1 + 2 + \ldots + n.
\]

1.6 SYMBOLIC EXPRESSIONS - SYMBOLS AND LIST

Exercise 1.6a. What do these expressions evaluate to?

\[
\begin{align*}
&\text{(setq a 123)} \\
&a \\
&\text{(quote a)} \\
&'a \\
&(+ a 17) \\
&'(+ 1 17) \\
&(setq a 'b) \\
&a \\
&'(quote x) \\
&''x \\
&"Kalle Pettersson" \\
&'Kalle\ Pettersson \\
&'|Kalle Pettersson|
\end{align*}
\]

1.7 BASIC FUNCTIONS FOR LISTS

Exercise 1.7a. What do these expressions evaluate to?

\[
\begin{align*}
&\text{(first '(one two three))} \\
&\text{(car '(one two three))} \\
&\text{(rest (first '((one two) three four)))} \\
&\text{(cdr (car '((one two) three four)))} \\
&\text{(cdar '((one two) three four)))} \\
&\text{(second 'one two three))} \\
&\text{(cadr 'one two three))} \\
&\text{(cons '(eva lisa) 'karl sven))} \\
&\text{(list '(eva lisa) 'karl sven))} \\
&\text{(append '(eva lisa) 'karl sven))} \\
&\text{(remove 'sven 'eva sven lisa sven anna))} \\
&\text{(subst 'gittan 'birgitta 'eva birgitta lisa birgitta karin))} \\
&\text{(make-list 5)} \\
&\text{(make-list 5 :initial-element 'start)} \\
&\text{(copy-list '(a new list))} \\
&\text{(intersection '(a b c) 'x b z c))} \\
&\text{(union 'a b c) 'x b z c))} \\
&\text{(last 'karl adam nilsson))} \\
&\text{(butlast 'karl adam nilsson))} \\
&\text{(butlast '(a b c d e) 3)} \\
&\text{(nthcdr 2 'a b c d e))} \\
&\text{(nth 2 'a b c d e))} \\
&\text{(assoc 'tre '((ett . one) (två . two) (tre. three)))}
\end{align*}
\]
1.8 RECURSION OVER SEQUENCES (LISTS)

Here you may need some training.

Exercise 1.8a. Define a function `add-only-numbers` which adds the numbers on the top level in a list.

```
(add-only-numbers '(a 1 b (b 4) 2 3))
=> 6
```

Exercise 1.8b. Define a function `remove-vocals` which takes a list of letters (symbols) and returns a new list where the vocals are removed.

```
(remove-vocals '(b i r g i t t a))
=> (b r g t t)
```

Exercise 1.8c. Functions for sets. There are set functions in Lisp. Here we want to implement some of those functions. Define `my-union` (union is already in the system), `my-intersection` och `my-difference`. Define also an operation `add-an-element` which adds an element to a set and also a function `member?` which checks if an element is a member of a set. The set is here implemented as a list.

```
(setq students '(sven lisa peter anna karl))
(setq football '(lisa eva karl sven))
(setq theater '(karin lisa peter nisse eva))
(setq all-persons (my-union students (my-union football theater)))
```

Which students can play football and be a team with the goal keeper Ravelli.

```
(add-a-member 'ravelli (my-intersection students football))
```

Which students have an interest?

```
(my-intersection students (my-union football theater))
```

1.9 DOUBLE RECURSION

Exercise 1.9a. Do exercise 1.8a but add all numbers on the list (regardless of level), call it `add-all-numbers`.

```
(add-all-numbers '(a 1 b (b 4) 2 3))
=> 10
```

Exercise 1.9b. Do the same for exercise 1.8b.

```
(remove-all-vocals '((v a d) (h e t e r) ((d i n) (b i l))))
=> ((v d) (h t r) ((d n) (b l)))
```

1.10 RECURSION OVER BINARY TREES

The list can be seen as a binary tree and therefore be processed as one.

Exercise 1.10a. Write the list `(1 (2 3))` in dotted pair notation. Give the list in both notations to the Lisp system and look at the results.

What will be printed by

```
(cons (cons 'a 'b) (cons 'c nil))
```

Exercise 1.10b. We see the list as a binary tree and can therefor contain dotted pairs. Write a function `add-number-leaves` which adds all number leaves.

```
(add-number-leaves '(((a . 1) . 3) . (c . 4)))
=> 8
```
1.11 FORMULAS - LOGICAL EXPRESSIONS

Formulas can easily be represented as trees and be implemented in lists. The formulas are processed with binary tree recursion.

**Exercise 1.11.a.** Define the logical operations disjunction ($\land$), conjunction ($\lor$) and implication ($\rightarrow$) by using if. Give the new functions the name my-or, my-and and my-implication.

```lisp
(my-or t nil)
=> t

(my-and nil t)
=> nil

(my-implication nil t)
=> t
```

**Exercise 1.11.b.** Prefix -> infix notation. Define a function transform which transforms an expression from prefix notation to infix notation. All operations are binary.

```lisp
(transform ’10)
=> 10

(transform ’(+ 10 7))
=> (10 + 7)

(transform ’(* (+ x) (- 20 (* 2 y))))
=> ((10 + x) * (20 - (2 * y)))
```

**Exercise 1.11.c.** Use Lisp to show that the following propositional formulas are tautologies (they are always true).

(p $\land$ (p $\rightarrow$ q)) $\rightarrow$ q
((p $\rightarrow$ q) $\land$ $\neg$q) $\rightarrow$ $\neg$p
ASSIGNMENT 1A - EVALUATION OF LOGICAL EXPRESSIONS

Define a function \texttt{logical-value} which takes two arguments, a propositional formula and a table with truth values (true or false) for each propositional variable, and calculates the value of the logical formula.

We represent the truth values with the Lisp symbols \texttt{true} and \texttt{false}. You can read more about propositional calculus in Russel-Norvig’s book Artificial intelligence - A Modern Approach i section 6.4.

In exercise 1.11a you defined your own logical operator. Redefine them to work on Lisp symbols \texttt{true} and \texttt{false} instead of \texttt{t} and \texttt{nil}.

We can store the propositional variables with their values in an association list. Example:

\[
((a \ . \ true) \ (b \ . \ false) \ (c \ . \ true))
\]

A logical expression \texttt{log-expr} is defined as:

- "logical constants", \texttt{true} or \texttt{false}
- "propositional symbols" (a, b, c, p1, p2, ...)
- (not \texttt{log-expr})
- (\texttt{log-expr} and \texttt{log-expr})
- (\texttt{log-expr} or \texttt{log-expr})
- (\texttt{log-expr} implies \texttt{log-expr})

\[
\begin{align*}
\text{logical-value} \ 'a' \ ((a \ . \ true)) & \Rightarrow \text{true} \\
\text{logical-value} \ '(a \ or \ (not \ a)) \ ((a \ . \ false)) & \Rightarrow \text{true} \\
\text{logical-value} \ '((p1 \ and \ p2) \ and \ (not \ (p3 \ or \ false))) \ ((p1 \ . \ true) \ (p2 \ . \ false) \ (p3 \ . \ true)) & \Rightarrow \text{false}
\end{align*}
\]

ASSIGNMENT 1B - FIND OCH BACKFIND

We want two functions to find a new position in a list. Compare with an editor where we want to find the first or the last occurrence of a symbol and move to that position. If the symbol does not exist we want to stay where we are.

The function \texttt{ed-find} should find the first occurrence of a symbol in an arbitrary list and return the rest of the list (or sublist) where the actual symbol is the first element. The function \texttt{backfind} finds the last occurrence on an arbitrary list. If the symbol is not found the original list is returned.

\[
\begin{align*}
\text{ed-find} \ 'y' \ ((a \ b) \ c \ y \ d)) & \Rightarrow \ (y \ d) \\
\text{ed-find} \ 'x' \ ((a \ x \ b \ x \ c) \ x \ d)) & \Rightarrow \ (x \ b \ x \ c) \\
\text{ed-find} \ 'x' \ ((a \ b \ c)) & \Rightarrow \ ((a \ b) \ c) \\
\text{backfind} \ 'x' \ ((a \ (x \ x)) \ x \ c \ x \ d \ e) & \Rightarrow \ (x \ d \ e) \\
\text{backfind} \ 'z' \ ((a \ z) \ c \ d)) & \Rightarrow \ (z) \\
\text{backfind} \ 'x' \ ((a \ b) \ c)) & \Rightarrow \ ((a \ b) \ c)
\end{align*}
\]

; only one occurrence

; more occurrences, the first is chosen

; the symbol does not occur in the list

; more occurrences, the last is chosen

; only one occurrences

; the symbol does not occur in the list