LUND UNIVERSITY

LISP -- A One-Week Course

Årzén, Karl-Erik

1986

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Årzén, K.-E. (1986). *LISP -- A One-Week Course*. (Technical Reports TFRT-7310). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

CODEN: LUTFD2/(TFRT-7310)/1-144/(1986)

LISP - a one-week course

Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
January 1986

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
January 1986

Document Number
CODEN: LUTFD2/(TFRT-7310)/1-144/(1986)

Author(s)
Karl-Erik Årzén

Supervisor

Sponsoring organisation

Title and subtitle
LISP — a one-week course.

Abstract

This report contains the material handed out during a one week course in Lisp given at the Department of Automatic Control Dec. 19th 1985 – Jan. 10th 1986.

The material consists of copies of the viewgraphs, exercises with solutions and project suggestions. The course has been hold as close to Common Lisp as possible with some minor Franz Lisp extensions.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
144

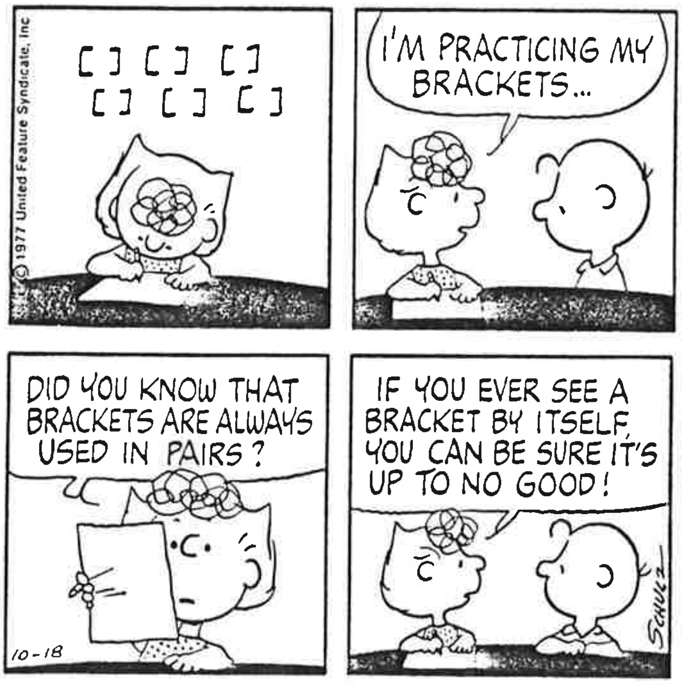
Security classification

Recipient's notes

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

CONTENTS

LISP



© 1977 United Feature Syndicate, Inc

1. INTRODUCTION

- Motivation
- Interpretation and evaluation
- List functions
- Predicates
- Function definitions
- Conditionals
- Recursion

LISP

LISt Programming

The most used language for symbolic processing.

Computing with representations of information inside a computer that are closer to the way a programmer, or the person specifying the problem, thinks about the problem than those representations used in Fortran-style "computing with numbers".

Why Lisp?

- The interaction argument. Oriented toward programming at a terminal with rapid response.

- The environment argument. Sophisticated computing environment makes it possible to write big, complicated programs.

- The features argument. Lisp was designed for symbol processing and has been developed in that direction.

- The uniformity argument. Lisp procedures and Lisp data have the same form.

Myths

- Lisp is slow at arithmetics.
- Lisp is slow.
- Lisp programs are big.
- Lisp is hard to read and debug because of all the parentheses.
- Lisp is hard to learn.

The interpreter

The interpreter works in a **read-eval-print** loop.

```
->(+ 8 3)
11
```

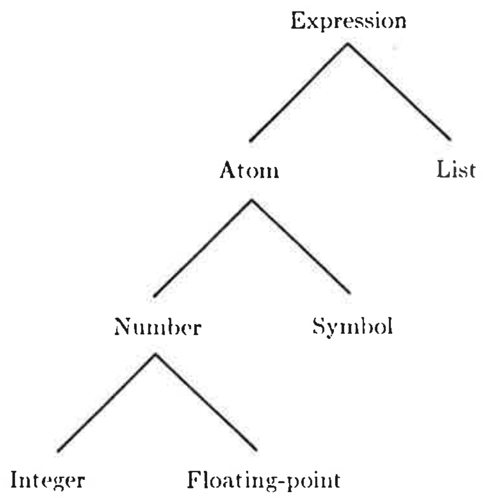
```
->(max 2 4 3)
4
```

Lisp evaluates everything.

```
->8
8
```

Expressions always return a value.

Basic data objects



S-expression: Symbolic expression as opposed to Meta expressions used by John McCarty.

Form: An expression is called a form if it is meant to be evaluated. If it is a list, the first element generally is the name of a procedure that is used in the evaluation process.

Binding

Atoms play the role of variables.

Binding is another name for assigning a value. The functions is called **setq**.

```
->(setq x 5)
5
->x
5
->(plus x 8)
13
->y
Error: Unbound variable: y
<1>:
<1>: (reset)
->
```

When an atom is bound to a value with **setq** this value is returned as the value of setq. As a side-effect the atom is bound to the value.

Reset is used to return to the top-level from an error. Using Ctrl-D or Ctrl-Z is equivalent.

Quoting

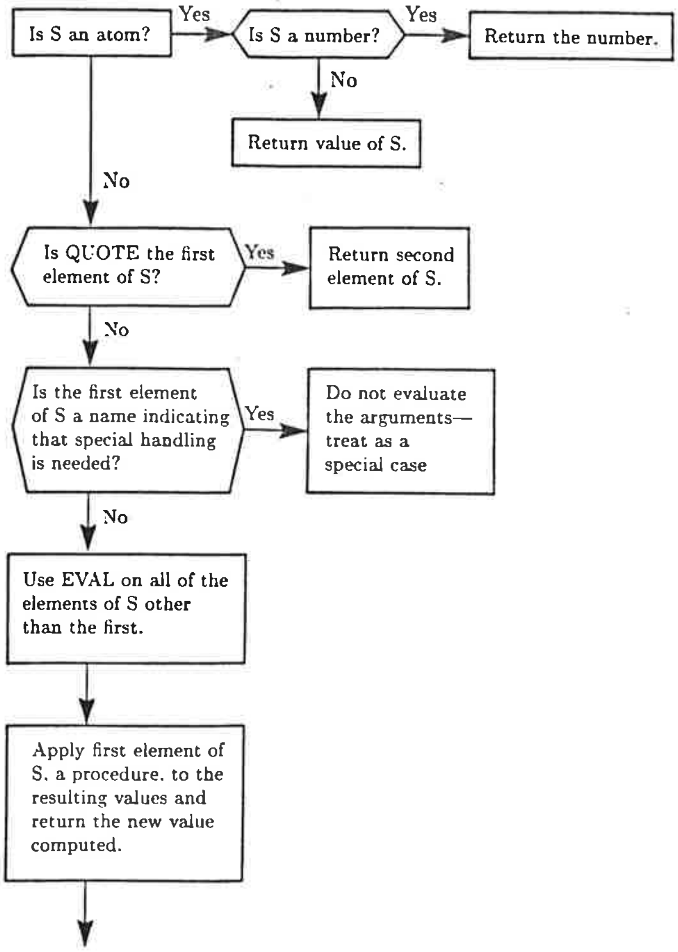
Evaluation can be stopped by quoting.

```
->(setq x 5)
5
->x
5
->'x
x
->(setq y '(a b c))
(a b c)
```

`'x` is *syntactic sugar* for `(quote x)`.

Lisp distinguishes between the *name* of an atom and its *value*.

Evaluation



Special forms are treated specially by the evaluator. An example is **setq**

Eval can be used explicitly. It then causes an extra evaluation.

```
->(setq a 'b)
b
->(setq b 'c)
c
->(eval a)
c
```

Arithmetics

Always prefix notation.

Add, plus, sum, + returns the sum of the arguments. + takes only fixnums.

Diff, difference, - subtracts from the first argument all subsequent arguments. - for fixnums.

Times, product, * returns the product of all the arguments. * for fixnums.

Quotient, / divides the first argument by succeeding ones.

Sin, cos, asin, atan, abs, expt, log, max, sqrt ...

Very large numbers possible.

Taking lists apart

Car returns the first element of a list.

```
->(car '(a b c))
a
->(car '((a b)))
(a b)
```

Cdr returns the list with its first element missing.

```
->(cdr '(a b c))
(b c)
```

Car and **cdr** are non-destructive.

```
->(setq x '(a b c))
(a b c)
->(cdr x)
(b c)
->x
(a b c)
```

The names come from the original IBM 704 implementation: Contents of address or decrement register.

Car and **cdr** can be composed.

```
(car (cdr x)) = (cadr x)
(cdr (car x)) = (cdar x)
(car (car x)) = (caar x)
```

```
->(cadr '((a b c) (d e f)))
(d e f)
->(cadadr '((a b c) (d e f)))
e
```

List construction

Cons adds a new element to the front of a list. It is the inverse of car and cdr.

```
->(cons 'a '(b c))
(a b c)
->(setq x '(a b c))
(a b c)
->(cons (car x) (cdr x))
(a b c)
```

Append takes any number of lists and creates a new list by sticking them together.

List makes a list out of its arguments.

```
->(append '(a b) '(c d) '(e f))
(a b c d e f)
->(list '(a b) '(c d) 'e)
((a b) (c d) e)
```

List functions

Length returns the length of a list.

Reverse turns the top-level of a list around.

```
->(length (append '(1 2) '(3 4)))
4
->(reverse '(a b c))
(c b a)
```

Subst substitutes all occurences of an atom in a list for a new expression.

Last returns a list that contains the last element of the list given as argument.

```
->(subst 'a 'b '(a b (b c)))
(a a (a c))
->(last '(a b c))
(c)
```

Predicates

A predicate is a functions that returns either **t** or **nil.** Everything that is not nil is considered to be true. The empty list is denoted (). It is treated the same way as nil.

Type predicates

In conventionnal programming langauages types are associated with variables and the type checking is performed during compile time. In Lisp, types are associated with Lisp data objects and the type checking is performed during run-time.

Atom returns t if the argument is an atom.

Listp returns t if the argument is a list.

```
->(atom 'a)
t
->(atom 3)
t
->(listp 'a)
nil
->(listp '(a b c))
```

```
t
->(atom nil)
t
->(listp nil)
t
```

Equal tests if two expressions look alike.

Null returns t if its argument evaluates to t.

```
->(equal 'a 'b)
nil
->(setq a 5)
5
->(equal a 5)
t
->(null '())
t
```

Member tests if the first argument is a member of the top-level of the second. If so it returns the list beginning with this argument.

```
->(member 'b '(a b c))
(b c)
->(member '(a b) '(a b c))
nil
->(member 'y '(x (y) z))
nil
```

Numberp, zerop, evenp, oddp, greaterp, lessp, minusp do what you would expect.

Defining functions

Procedure abstraction is the process of constructing new procedures by combining existing ones.

```
(defun <procedure name>
   (<parameter 1> ... <parameter n>)
   <procedure body>)
```

```
-> (defun cube (x)
       (times x x x))
cube
-> (cube 2)
8
```

The evaluation of `(cube 2)` starts with the evaluation of the argument 2. Then the formal parameters are bound to the value of the corresponding actual parameters. Next, Lisp evaluates each form in the procedure body and returns the value of the last. Also, the previous values of the formal parameters are restored.

A normal Lisp function is *called by value. Call by reference* is impossible. All output values from a function are returned as the result of the function. It is impossible to return results through the arguments.

Conditionals

The standard conditional function in Lisp is the **cond**. The syntax looks like

```
(cond
   (<test 1> ... <result 1>)
   (<test 2> ... <result 2>)
     .
     .
   (<test n> ... <result n>))
```

The arguments to are called **cond-clauses**. Each **cond-clause** consists of a series of expressions. The first of the expressions is treated as a test and the rest is the things to do if the test is t. Lisp returns the value of the last expression in the first **cond-clause** which test is t. If no successful clause is found **cond** returns nil.

```
(cond ((null l) 'empty)
      (t 'not-empty))
```

```
(cond (l 'not-empty)
      (t 'empty)
```

```
-> (defun enter (e l)
     (cond ((member e l) l)
           (t (cons e l))))
```

If does also exist. This is expanded to a **cond**.

```
(if a then b c
      elseif d then e f
      elseif g thenret
      else h i)
```

Logical operators

Not returns t if its argument evaluates to nil. The same as null.

And takes any number of arguments, which are evaluated one after each other. The evaluation goes on until some argument evaluates to nil in which case **and** returns nil. If it reaches the end without any nil then **and** returns the value of its last argument which is guaranteed non-nil.

Or works similarly. It stops before the end only if some argument evaluates to true. If any does, **or** returns the value of that argument; otherwise it returns nil.

Cond can be used instead of both **and** and **or**. The code, however becomes difficult to read.

Recursion

Lisp is built around recursion.

Example: Factorial

```
->(defun factorial (n)
     (cond ((equal n 0) 1)
           (t (times n (factorial (sub1 n))))))
```

Algorithmically this is called a *linearly recursive process.*

```
->(defun factorial (n)
    (fact-iter 1 1 n))

->(defun fact-iter (product counter max-count)
    (cond ((greaterp counter max-count) product)
          (t (fact-iter (times counter product)
                        (add1 counter)
                        maxcount))))
```

Algorithmically this is not a recursion. It is an *iterative process.* It is also called *tail recursion.* Some Lisp interpreters and most Lisp compilers detect this and do not generate new stack entries.

Tree recursion

Example: Fibonacci.

```
->(defun fib (n)
     (cond ((equal n 0) 0)
           ((equal n 1) 1)
           (t (plus (fib (- n 1))
                    (fib (- n 2))))))
```

Much redundant computation.

```
->(defun fib (n)
    (fib-iter 1 0 n))

->(defun fib-iter (a b count)
    (cond ((equal 0 count) b)
          (t (fib-iter (+ a b)
                       a
                       (sub1 count)))))
```

List Recursion

The most common form of recursion in Lisp.

Example: Increment all the elements in a list with 1.

```
->(defun increment (list)
    (cond ((null list) nil)
          (t (cons (add1 (car list))
                   (increment (cdr list))))))
```

Example: Member

```
->(defun member (e l)
    (cond ((null l) nil)
          ((equal e (car l)) l)
          (t (member e (cdr l)))))
```

2. ENVIRONMENT

- Loading files

- Editing and Emacs

- Debugging

Using Franz Lisp

Do the following definitions in your login.com.

```
$ lisp:== $eun_root:[usr.ucb]lisp /usr/ucb/lisp
$ emacs:== @scr:[karlerik.slask.anders.newemacs]emacs
```

Lisp is started by typing **$ lisp**.

Lisp starts with loading the file **lisprc.l** from your home directory. This file allows you to set up your own defaults, read in files etc. This file should contain the following forms.

```
(load 'dcl)
(load 'functions)
```

These files contains some utility functions and they are loaded from the lisp library directory, /usr/lib/lisp. **Load** is the standard way to load in files into Lisp. It first searches for the file in the current directory and then in /usr/lib/lisp. When no file extension is given it first searches for .o files and then for .l files. Compiled lisp files have the extension .o.

Utility Functions

Functions returns a list of user defined, non-compiled Lisp functions.

Dcl makes it possible to do most DCL commands from inside Lisp. Another possibility is to do Ctarl-Y, spawn, your own command, logout and continue.

Peve calls the Peve editor with the given filename. The file is loaded when Peve is left. This function does not evaluate its argument.

Emacs the same but with Emacs.

```
->(dcl 'print 'test1.l)

->(emacs test1.l)
```

Editing

It is a good habit to use a screen-oriented text editor when you write your programs. The other possibility is to enter your functions from the top-level, edit them with Lisp's expression-oriented structure editor and to write them out with the pp (pretty-print) function. The editor is however very difficult to use and I don't recommend it.

The parenthesis may feel disturbing. Some rules to follow are.

- More than 8-9 parenthesis in a row means that you have'nt structured your program enough.
- Indentation.
- Use Emacs.
- Super-parenthesis.

EMACS

Emacs is a very powerful, extensible display editor. It provides features like several windows and buffers, user programmable keyboard and structure editing facilities. When Emacs is entered with a .l file it is automatically changing to Lisp-mode. Emacs is always in insert char mode.

Start by copying use:[karlerik]emacsinit.pro to your home directory.

Necessary commands.

The arrow keys work.

```
^L     Redisplay screen
^X-^F  Write out files and exit
LF     Indented Return
DEL    Delete left char
^A     Move to beginning of line
^E     Move to end of line
^K     Delete rest of line (^Y gets it back)
^Space Set the mark
^W     Delete region between mark and cursor
^Y     Yanks it pack at the cursor
^X-^X  Interchange cursor and mark
^s     Incremental search
```

Unix Emacs Reference Card

SOME NECESSARY NOTATION

	Any ordinary character goes into the buffer (no insert command needed). Commands are all control characters or other characters prefixed by Escape or a control-X. Escape is sometimes called Meta or Altmode in EMACS.
↑	A control character. ↑F means "control F".
ESC-	A two-character command sequence where the first character is Escape. ESC-F means "ESCAPE then F".
ESC-X string	A command designated "by hand". "ESC-x read-file" means: type "Escape", then "x", then "read-file", then (cr).
dot	EMACS term for cursor position in current buffer.
mark	An invisible set position in the buffer used by region commands.
region	The area of the buffer between the dot and mark.

CHARACTER OPERATIONS

↑B	Move left (Back)
↑F	Move right (Forward)
↑P	Move up (Previous)
↑N	Move down (Next)
↑D	Delete right
↑H or BS or DEL or RUBOUT	Delete left
↑T	Transpose previous 2 characters (ht_ -> th_)
↑Q	Literally inserts (quotes) the next character typed (e.g. ↑Q-↑L)
↑U-n	Provide a numeric argument of n to the command that follows (n defaults to 4, eg. try ↑U-↑N and ↑U-↑U-↑F)
↑M or CR	newline
↑J or NL	newline followed by an indent

WORD OPERATIONS

ESC-b	Move left (Back)
ESC-f	Move right (Forward)
ESC-d	Delete word right
ESC-h	Delete word left
ESC-c	Capitalize word
ESC-l	Lowercase word
ESC-u	Uppercase word
ESC-↑	Invert case of word

LINE OPERATIONS

↑A	Move to the beginning of the line
↑E	Move to the end of the line
↑O	Open up a line for typing
↑K	Kill from dot to end of line (↑Y yanks it back at dot)

PARAGRAPH OPERATIONS

ESC-[	Move to beginning of the paragraph
ESC-]	Move to end of the paragraph
ESC-j	Justify the current paragraph

GETTING OUT

↑X-↑S	Save the file being worked on
↑X-↑W	Write the current buffer into a file with a different name
↑X-↑M	Write out all modified files
↑X-↑F	Write out all modified files and exit
↑C or ESC-↑C or ↑X-↑C	Finish by exiting to the shell
↑_	Recursively push (escape) to a new shell

SCREEN AND SCREEN OPERATIONS

↑V	Show next screen page
ESC-V	Show previous screen page
↑L	Redisplay screen
↑Z	Scroll screen up
ESC-Z	Scroll screen down
ESC-!	Move the line dot is on to top of the screen
ESC-,	Move cursor to beginning of window
ESC-.	Move cursor to end of window
↑X-2	Split the current window in two windows (same buffer shown in each)
↑X-1	Resume single window (using current buffer)

↑X-d	Delete the current window, giving space to window below
↑X-n	Move cursor to next window
↑X-p	Move cursor to previous window
ESC-↑V	Display the next screen page in the other window
↑X-↑Z	Shrink window
↑X-z	Enlarge window

BUFFER AND FILE OPERATIONS

↑Y	Yank back the last thing killed (kill and delete are different)
↑X-↑V	Get a file into a buffer for editing
↑X-↑R	Read a file into current buffer, erasing old contents
↑X-↑I	Insert file at dot
↑X-↑O	Select a different buffer (it must already exist)
↑X-B	Select a different buffer (it need not pre-exist)
↑X-↑B	Display a list of available buffers
ESC-↑Y	Insert selected buffer at dot
ESC-<	Move to the top of the current buffer
ESC->	Move to the end of the current buffer

HELP AND HELPER FUNCTIONS

↑G	Abort anything at any time.
ESC-?	Show every command containing string (try ESC-? para)
ESC-X info	Browse through the Emacs manual.
↑X↑U	Undo the effects of previous commands.

SEARCH

↑S	Search forward
↑R	Search backward

REPLACE

ESC-r	Replace one string with another
ESC-q	Query Replace, one string with another

REGION OPERATIONS

↑@	Set the mark
↑X-↑X	Interchange dot and mark (i.e. go to the other end of the region)
↑W	Kill region (↑Y yanks it back at dot)

MACRO OPERATIONS

↑X-(	Start remembering keystrokes, ie. start defining a keyboard macro
↑X-)	Stop remembering keystrokes, ie. end the definition
↑X-e	Execute remembered keystrokes, ie. execute the keyboard macro

COMPILING (MAKE) OPERATIONS.

↑X-↑E	Execute the "make" (or other) command, saving output in a buffer
↑X-↑N	Go to the next error in the file
↑X-!	Execute the given command, saving output in a buffer

MAIL

↑X-r	Read mail.
↑X-m	Send mail

Trace

`(trace function)` starts tracing of this function.

`(untrace function)` stops tracing of this function.

Step

`(step t)` starts stepping of the evaluation.
`(step function)` starts the stepping when this function is evaluated.

```
<ret> Continue stepping
  c   Show returned value from this level
      and continue upwards
  g   Turn off stepping but continue
      evaluation
  q   Quit stepping
```

Break

When an error occurs the evaluation is stopped and Lisp enters a break loop. Here all variables can be inspected.

Lisp returns to top-level with `(reset)`, Ctrl-Z or Ctrl-D. The evaluation is instead continued if a value is returned with `(return value)`.

A break loop can also be entered by the command `(break)`.

Stack dumps

`(showstack)` prints the contents of the lisp evaluation stack in reverse order. This can be useful while searching for errors.

`(baktrace)` is a less verbose version of showstack.

`(debug)` can be used to enter the special Lisp debugger. This can be used to inspect and manipulate the lisp evaluation stack in different ways. Difficult.

3. FUNCTION DEFINITIONS

- Functions as arguments
- Anonymous functions
- Argument keywords
- Fexpr and Lexpr
- Scoping
- Functions as returned values
- Iteration

Functions as arguments

Ex: Compute the sum of the integers between a and b.

```
(defun sum-integers (a b)
  (cond ((greaterp a b) 0)
        (t (plus a (sum-integers (add1 a) b)))))
```

Ex: Compute the sum of the cubes of the integers.

```
(defun sum-cubes (a b)
  (cond ((greaterp a b) 0)
        (t (plus (cube a)
                 (sum-cubes (add1 a) b)))))
```

Instead

```
(defun sum-loop (term a next b)
  (cond ((greaterp a b) 0)
        (t (plus (term a)
                 (sum-loop term (next a) next b)))))

(defun sum-cubes (a b)
  (sum-loop 'cube a 'add1 b))
```

In standard Lisp a symbol has both a value and a function binding.

```
->(setq car '5)
5
->car
5
->(car '(a b c))
a
->(setq func 'car)
car
->(func '(a b c))
a
```

The functions binding of the first argument is used when a function call is evaluated. If the first argument is a symbol and has no functions binding then this symbol is evaluated and the returned value is used.

```
(defun sum-loop (term a next b)
 (cond ((greaterp a b) 0)
       (t (plus (funcall term a)
                (sum-loop term
                          (funcall next a)
                          next
                          b)))))
```

Apply takes two arguments, which are both evaluated. The first should evaluate to a function, and the second, to a list of arguments to that function.

Funcall works like apply except that it expects the arguments to its functions to appear one after another directly after the function argument.

```
->(apply 'cons '(a (b c d)))
(a b c d)
->(funcall 'cons 'a '(b c d))
(a b c d)
```

Anonymous functions

Sometimes it is desirable to define functions that have no name, e.g. when they are only used once.

This is done with *lambda notation.* The function is defined with a lambda form. The cube example looks like

`(lambda (x) (times x x x))` The lambda form is equivalent to a function.

```
->((lambda (x) (times x x x)) 3)
27

->(defun sum-cubes (a b)
    (sum-loop '(lambda (x) (times x x x)) a 'add1 b))
```

Def

The Lisp interpreter always works with lambda forms when it comes to function application. The def function for defining functions require the user to explicitly specify a lambda form.

```
(def cube (lambda (x) (times x x x)))

(defun cube (x) (times x x x))
```

Argument keywords

Variable number of arguments
Ex: My-plus

```
(defun my-plus (&rest list)
   (apply 'plus list))
```

Optional arguments
Ex: Power

```
(defun power (x &optional (y 2))
  (expt x y))
```

The parameter following **&optional** will be bound to the corresponding argument if one is supplied. If &optional is followed by a list `(name value)` then name is interpreted as the optional parameter and value is the default value.

&aux can be used to declare local variables. The symbol after **&aux** is treated as a local variable initialized to nil. If a list `(name expr)` is given then name is bound to expr.

The argument keywords can only be used with defun (or defmacro).

Fexpr and Lexpr

For historical reasons.

A *lexpr* is a function that takes any number of arguments and evaluates the arguments. The only formal parameter is bound to the number of arguments passed.

Ex: Print-no-of-arguments

```
(defun print-no-of-arguments n
  (patom "Number of arguments: ")
  (print n)
  (terpr))

->(print-no-of-arguments 'a 'b 'c)
Number of arguments: 3
nil


(def print-no-of-arguments
  (lexpr (n)
    etc.
```

Arg is used to reference to the individual arguments. It takes one argument that should evaluate to a number. `(arg 2)` returns the second argument.

Use **&rest** instead.

A *fexpr* is a function that takes any number or arguments none of which is evaluated. Functions that do not evaluate their arguments are sometimes useful at top-level.

Ex: consq

```
(defun consq fexpr (l)
 (cons (car l) (cadr l)))

->(consq a (b c d))
(a b c d)

(def consq
  (nlambda (l)
    (cons (car l) (cadr l))))
```

Use macros instead.

Let

Defines local variables

```
(let ((<par 1> <val 1>)
      (<par 2> <val 2>)
        .
      (<par n> <val n>))
  <exp 1> .. <exp n>)
```

The parameters are bound to their to initial values and the expressions in the let body are evaluated. The parameters are bound in parallel. Lambda notation can be used instead.

```
((lambda (par1 .. parn)
   exp1 .. expn) val1 .. valn)
```

Let* is the same as **let** except that the binding is sequential.

Scoping

A collection of bindings is called an *environment.* A symbol's value is found by looking in the appropriate environment.

Dynamic scoping means that the values of free variable are determined by the *activation environment*, the environment in force when the procedure requiring the free variable is called. Used in most interpreted old Lisps.

Lexical scoping means that the values of free variables are determined by the *definition environment*, the environment in force when the procedure requiring the free variable was defined. Used in Common Lisp, Scheme, compiled Franz Lisp and traditional languages.

Ex: Sum of powers. Dynamic scoping.

```
(defun sum-powers (a b n)
  (sum-loop 'nth-power a 'add1 b))

(defun nth-power (x)
  (expt x n))
```

Ex: Lexical scoping.

```
(defun sum-powers (a b n)
  (defun nth-power (x)
    (expt x n))
  (sum-loop 'nth-power a 'add1 'b))
```

Functions as returned values

In practice requires lexical scoping.

Common Lisp examples:

```
(defun compose (f g)
  #'(lambda (x) (funcall f (funcall g x))))

->(funcall (compose #'sqrt #'abs) -9.0)
3.0

(defun deriv (f dx)
 #'(lambda (x)
     (/ (- (funcall f (+ x dx)) (funcall f x))
        dx)))

->(funcall (deriv #'cube 0.001) 5)
75.15
```

Iteration

Dealing with lists often calls for iteration.

Mapcar maps its first element which should evaluate to a function over the elements of second argument.

```
->(mapcar 'add1 '(1 2 3 4))
(2 3 4 5)
->(mapcar 'oddp '(1 2 3 4))
(t nil t nil)
->(mapcar 'plus '(1 2 3 4) '(10 20 30 40))
(11 22 33 44)
```

Ex: Sum-of-squares

```
(defun sum-of-squares (&rest list)
  (apply 'plus
         (mapcar 'sqr list)))
```

Can be written with list recursion.

Example: Compute the number of atoms in a list structure.

```
(defun count-atoms (list)
   (cond ((null list) 0)
         ((atom list) 1)
         (t (apply 'plus
                   (mapcar 'count-atoms
                           list)))))

->(count-atoms '(1 2 (a b (c)) (3 4)))
7
```

Do

Binds parameters and supports explicit iteration.

```
(do ((var1 initval1 repval1)
     (var2 initval2 repval2)
     .
     (varn initvaln repvaln))
 <termination-test>
 exp1 ... expn)
```

Ex: Our-expt and our-reverse.

```
(defun our-expt (m n)
  (do ((result 1)
       (exponent n))
    ((zerop exponent) result)
       (setq result (times m result))
       (setq exponent (sub1 exponent))))

(defun our-reverse (list)
  (do ((x list (cdr x))
       (res nil (cons (car x) res)))
    ((null x) res)))
```

Prog

Old-fashioned do with goto and return. Should be avoided.

```
(prog (var1 .. varn)
  exp1 .. expn)
```

Ex: Our-expt

```
(defun our-expt (m n)
  (prog (result exponent)
    (setq result 1)
    (setq exponent n)
 loop
    (cond ((zerop exponent)
           (return result)))
    (setq result (* m result))
    (setq exponent (sub1 exponent))
    (go loop)))
```

4. DATA ABSTRACTION

- Association lists
- Property lists
- Data abstraction
- Rational arithmetics example
- Symbolic differentiation example
- Read and Write

Association lists

An *association list* is a list of sublists, in which the first element of each sublist is a *key*.

```
->(setq brick-a '((color red)
                  (supported-by brick-b)
                  (is-a brick)))
((color red) (supported-by brick-b) (is-a brick))
```

Assoc is used to retrieve values.

```
->(assoc 'color brick-a)
(color red)
->(assoc 'is-a brick-a)
(is-a brick)
```

Property lists

Symbols can have *properties.* To describe an object, we need *property names* and *property values.* The properties are stored on a *property list* associated with each symbol.

Putprop is used to assign properties to symbols.

Get is used to access the stored values.

Plist returns the property list of a symbol.

```
->(putprop 'chair3 'blue 'color)
blue
->(putprop 'chair3 'john 'owner)
john
->(get 'chair3 'color)
blue
->(plist 'chair3)
(color blue owner john)
```

Ex: Database of information about books in the library. The global variable **Library** is used to hold the list of all books we know about.

```
(defun add-book (bookref title author publisher)
  (putprop bookref title 'title)
  (putprop bookref author 'author)
  (putprop bookref publisher 'publisher)
  (setq library (cons bookref library))
  bookref)

->(setq library nil)
nil
->(add-book 'book1
            '(War and Peace)
            '(Leo Tolstoy)
            '(Press Int))
book1
->(add-book 'book2
            '(Artificial Intelligence)
            '(Patrick Winston)
            '(Addison-Wesley))
book2
->(add-book 'book3
            '(Data structure techniques)
            '(Tim Standish)
            '(Addison-Wesley))
book3
```

```
(defun retrieve-by (property value)
  (remove-if-not '(lambda (x)
                     (equal (get x property)
                            value))
                  library))

->(retrieve-by 'author '(Leo Tolstoy))
(book1)
->(retrive-by 'publisher '(Addison-Wesley))
(book2 book3)
```

Data Abstraction

Data abstraction enables us to isolate how a compound data is used from the details of how it is constructed from more primitive data objects.

The interface between the abstract data objects and the actual data representation is a set of procedures, called *constructors*, *selectors* and *mutators*.

Constructors create abstract data objects.

Selectors access these data objects.

Mutators make changes to them.

Together they are called *access procedures*.

Ex: Arithmetic operators for rational numbers

Make-rat takes two integers n and d and returns the rational number whose numerator is n and whose denominator is d.

Numer takes a rational naumber and returns its numerator.

Denom takes a rational number and returns its denominator.

```
(defun +rat (x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(defun -rat (x y)
  (make-rat (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y))))

(defun *rat (x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(defun /rat (x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))

(defun =rat (x y)
```

```
  (equal (* (numer x) (denom y))
         (* (numer y) (denom x))))
```

Rational numbers implemented as a list of two elements.

```
(defun make-rat (n d)
  (list n d))

(defun numer (x) (car x))
(defun denom (x) (cadr x))

(defun print-rat (x)
  (princ (numer x))
  (princ "/")
  (princ (denom x))
  (terpr))

->(setq one-half (make-rat 1 2))
(1 2)
->(print-rat one-half)
1/2
nil
->(setq one-third (make-rat 1 3))
(1 3)
->(print-rat (+rat one-third one-third))
6/9
nil

(defun make-rat (n d)
  (let ((g (gcd n d)))
    (list (/ n g) (/ d g))))

(defun gcd (x y)
```

```
  (cond ((zerop y) x)
        (t (gcd y (remainder x y)))))

->(print-rat (+rat one-third one-third))
2/3
nil
```

Ex: Symbolic derivation.

```
(defun deriv (exp var)
  (cond ((constant? exp var) 0)
        ((same-var? exp var) 1)
        ((sum? exp)
         (make-sum (deriv (addend exp)
                          var)
                   (deriv (augend exp)
                          var)))
        ((product? exp)
         (make-sum
           (make-product
             (multiplier exp)
             (deriv (multiplicand exp)
                    var))
           (make-product
             (deriv (multiplier exp)
                    var)
             (multiplicand exp))))
         ;; More rules
         ))
```

Expressions implemented as list structures.

```
(defun constant? (exp var)
  (and (atom exp)
       (not (equal exp var))))

(defun same-var? (exp var)
  (and (atom exp)
       (equal exp var)))
```

```lisp
(defun sum? (exp)
  (and (listp exp)
       (equal (car exp) '+)))

(defun addend (exp) (cadr exp))
(defun augend (exp) (caddr exp))

(defun make-sum (addend augend)
  (list '+ addend augend))

(defun product? (exp)
  (and (listp exp)
       (equal (car exp) '*)))

(defun multiplier (exp) (cadr exp))
(defun multiplicand (exp) (caddr exp))

(defun make-product (multiplier multiplicand)
  (list '* multiplier multiplicand))


(setq expr '(+ (* a (* x x))  ;a*x*x + b*x + c
               (+ (* b x)
                  c)))

->(deriv expr 'x)
(+ (+ (* a (+ (* x 1) (* 1 x))) ; 2a*x + b
      (* 0 (* x x)))
  (+ (+ (* b 1) (* 0 x))
     0))
```

Add simplifications.

```
(defun constant? (exp var)
  (or (and (atom exp)
           (not (equal exp var)))
      (and (listp exp)
           (constant? (operand-1 exp) var)
           (constant? (operand-2 exp) var))))

(defun operand-1 (exp) (cadr exp))
(defun operand-2 (exp) (caddr exp))

(defun make-sum (a1 a2)
  (cond ((zerop a1) a2)
        ((zerop a2) a1)
        ((numberp a1)
         (cond ((numberp a2) (plus a1 a2))
               (t (make-sum-1 a1 a2))))
        ((numberp a2) (make-sum-1 a2 a1))
        (t (make-sum-1 a1 a2))))

(defun make-sum-1 (a1 a2)
  (list '+ a1 a2))

(defun make-product (a1 a2)
  (cond ((zerop a1) 0)
        ((zerop a2) 0)
        ((onep a1) a2)
        ((onep a2) a1)
        ((numberp a1)
         (cond ((numberp a2) (times a1 a2))
               (t (make-product-1 a1 a2))))
        ((numberp a2) (make-product-1 a2 a1))
        (t (make-product-1 a1 a2))))

(defun make-product-1 (a1 a2)
```

```
  (list '* a1 a2))

->(deriv expr 'x)
(+ (* a (+ x x)) b)
->(deriv expr 'a)
(* x x)
->(deriv expr 'b)
x
->(deriv expr 'c)
1
```

Read and Write

Read is a function of no arguments that causes Lisp to wait for a s-expression being typed in. This expression is returned.

Print prints its argument on the terminal and returns the value nil.

Terpr starts a new line.

The *escape character* \ allows the following character to escape from its normal Lisp interpretation. This means e.g. that parentheses can be used in symbol names. If more than one escape character is needed in a name it is possible to instead embed the name in vertical bars, |. These are sometimes called *symbol delimiters*.

Print prints out symbol names surrounded with vertical bars when needed.

Patom or **princ** prints out symbol names without vertical bars. Patom returns its argument and princ returns t.

```
->'ab\(cd
| ab(cd|
->'| ab(cd|
| ab(cd|
->(print 'ab\(cd)
| ab(cd| nil
->(princ '| a b c d| )
a b c dt
```

More on I/O in section 3 in the Appendix.

5. MACROS

- Macros
- Read macros and Backquote
- Internal representation
- Strings
- Arrays

Macros

Macros expands into a Lisp form which is then evaluated.

Macros do not evaluate their arguments.

```
(defmacro demo-macro (par)
  (patom par))

(defun demo-fun (par)
  (patom par))

->(setq this 'value-of-this)
value-of-this
->(demo-macro this)
thisvalue-of-this
->(demo-fun this)
value-of-thisvalue-of-this
```

&rest, **&optional** and **&aux** are allowed.

Suppose you often use cond in the following way:

```
(cond (<test> <result if success>)
      (t <result if failure>))
```

You might then want to define a function our-if that behaves in this way.

```
(our-if <test> <result-if-success>
               <result-if-failure>)

(defun our-if (test success failure)
  (cond (test success)
        (t failure)))

->(our-if (atom x) x (car x))  ;Bugged

(defmacro our-if (test success failure)
  (list 'cond (list test success)
              (list 't failure)))

->(macroexpand '(our-if (atom x) x (car x)))
(cond ((atom x) x) (t (car x)))
```

Read Macros

Through *read macros* the user can designate special characters to behave in unusual ways.

Suppose that the special character ' did not exist. We could then attach the following function to the ' character.

```
(lambda () (list (quote quote) (read)))
```

Typing `'a` would then result in `(quote a)`. The expansion is performed during the **read** phase of the **read-eval-print** loop.

The user can define new read macros.

Backquote

The backquote ‘ behaves in the same way as the normal quote ’ except that any commas that appear within the scope of the backquote have the effect of *unquoting* the following expression.

```
->(setq variable 'example)
example
->'(This is an variable)
(This is an variable)
->`(This is an ,variable)
(This is an example)
```

If an expression within the backquote is preceded by ,@ then the value of the expression is spliced into the list rather than inserted into it.

```
->(setq a '(1 2 3))
(1 2 3)
->`(,a b c)
((1 2 3) b c)
->`(,@a b c)
(1 2 3 b c)

->'`(,@a b c)
(append a '(b c))
```

The backquote is very useful in macro definitions.

```
(defmacro our-if (test success failure)
  `(cond (,test ,success) (t ,failure)))

(defmacro pop (stack)
 `(prog1
    (car ,stack)
    (setq ,stack (cdr ,stack))))

(defmacro push (element stack)
  `(setq ,stack (cons ,element ,stack)))

(defmacro my-load (file)
  `(load (quote ,file)))
```

Def for macros

Franz Lisp has two other ways to define a macro.

```
(defun our-if macro (arg)
 `(cond (,(cadr arg) ,(caddr arg))
        (t ,(cadddr arg))))

(def our-if (macro (arg)
 `(cond (,(cadr arg) ,(caddr arg))
        (t ,(cadddr arg)))))
```

The single formal parameter is bound to the *entire s-expression.*

Dotted pairs

Lists are internally represented using *dotted pairs.*

A dotted pair or a *cons cell* is a data structure with two entries; the car and the cdr pointer.

Lists are represented as binary trees of dotted pairs.

```
->(setq x '(a b c))
(a b c)
```

```
->'(a . nil)
(a)
->(car '(a . b))
a
->(cdr '(a . b))
b
->(cons 'a 'b)
(a . b)
```

The list construction functions you have seen before they all create new cons cells.

Nconc behaves like append but alter the memory cell contents.

```
->(setq x '(a b c))
(a b c)
->(setq y '(d e f))
(d e f)
->(setq z (append x y))
(a b c d e f)
->x
(a b c)
->(setq w (nconc x y))
(a b c d e f)
->x
(a b c d e f)
```

Rplaca takes two arguments, the first of which must be a list. It alters the list by replacing the car pointer of its first cell by a pointer to the second argument.

Rplacd behaves like rplaca but manipulates instead the cdr pointer.

```
->(setq x '(a b c))
(a b c)
->(rplaca x 1)
(1 b c)
->x
(1 b c)
->(rplacd x '(2 3))
(1 2 3)
->x
(1 2 3)
```

Eq is used to test if two structures really are the same. It returns t if the two arguments evaluate to the same internal Lisp pointer.

```
->(setq y (cdr x))
(2 3)
->(equal (cdr x) y)
t
->(eq (cdr x) y)
t
->(setq z '(2 3))
```

```
(2 3)
->(equal y z)
t
->(eq y z)
nil
```

Garbage Collection

```
->(setq x '(1 2 3))
(1 2 3)
->(setq x '(a b c))
(a b c)
```

The unused cons cells must be returned to the free storage list so they can be used again. They can not be immediately returned because other structures may be pointing to them. The *garbage collection* is performed when the system runs out of space.

Typical garbage collectors work with a *mark-sweep* algorithm. During the mark phase all used structures are marked and during the sweep phase the unmarked structures are returned.

Strings

A sequence of characters surrounded by double quotes.

Concat concatenates the values of its arguments into a new atom name. It accepts both atoms and strings.

Explode returns the list of characters that print would use to print the argument.

Get_pname returns the print name of an atom.

Substring returns a substring of a string.

```
->(concat 'abc "xyz")
abcxyz
->(explode 'abc)
(a b c)
->(get_pname 'abc)
"abc"
->(substring "abcdefghij" 5)
"efghij"
```

6. OBJECT-ORIENTED PROGRAMMING

- Complex arithmetics example
- Data-directed programming
- Message passing
- Flavors

Ex: Complex Arithmetic

```
(defun +c (z1 z2)
  (make-rectangular
     (plus (real-part z1) (real-part z2))
     (plus (imag-part z1) (imag-part z2))))

(defun -c (z1 z2)
  (make-rectangular
     (diff (real-part z1) (real-part z2))
     (diff (imag-part z1) (imag-part z2))))

(defun *c (z1 z2)
  (make-polar
     (times (magnitude z1) (magnitude z2))
     (plus (angle z1) (angle z2))))

(defun /c (z1 z2)
  (make-polar
     (quotient (magnitude z1) (magnitude z2))
     (diff (angle z1) (angle z2))))
```

Complex numbers can be represented in rectangular or polar form.

```
(defun make-rectangular (x y) (list x y))
(defun real-part (z) (car z))
(defun imag-part (z) (cadr z))

(defun make-polar (r a)
```

```
  (list (times r (cos a)) (times r (sin a))))

(defun magnitude (z)
  (sqrt (plus (square (car z)) (square (cadr z)))))

(defun angle (z)
  (atan (cadr z) (car z)))
```

or

```
(defun make-rectangular (x y)
 (list (sqrt (plus (square x) (square y)))
       (atan y x)))

(defun real-part (z)
 (times (car z) (cos (cadr z))))

(defun imag-part (z)
 (times (car z) (sin (cadr z))))

(defun make-polar (r a) (list r a))
(defun magnitude (z) (car z))
(defun angle (z) (cadr z))
```

Both representations. A type *rectangular* or *polar* is associated with each number.

```
(defun attach-type (type contents)
  (cons type contents))

(defun type (datum)
  (cond ((not (atom datum)) (car datum))
        (t (error "Bad typed datum " datum))))

(defun contents (datum)
  (cond ((not (atom datum)) (cdr datum))
```

```
        (t (error "Bad typed datum " datum))))

(defun polar? (z)
  (equal (type z) 'polar))

(defun rectangular? (z)
  (equal (type z) 'rectangular))

(defun make-rectangular (x y)
  (attach-type 'rectangular (list x y)))

(defun make-polar (r a)
  (attach-type 'polar (list r a)))
```

Now

```
(defun real-part (z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))))
```

`imag-part`, `magnitude`, `angle` in the same way.

```
(defun real-part-rectangular (z) (car z))
(defun real-part-polar (z)
  (times (car z) (cos (cadr z))))
```

`imag-part`, `magnitude`, `angle` divided in the same way.

Data-directed programming

A weakness is that the generic interface procedures `real-part, imagpart` ... must know all the different complex number representations.

Two-dimensional table.

Represent the table explicitly.

```
(putprop 'rectangular 'real-part-rectangular
                      'real-part)
(putprop 'rectangular 'imag-part-rectangular
                      'imag-part)
(putprop 'rectangular 'magnitude-rectangular
                      'magnitude)
(putprop 'rectangular 'angle-rectangular
                      'angle)
(putprop 'polar 'real-part-polar 'real-part)
(putprop 'polar 'imag-part-polar 'imag-part)
(putprop 'polar 'magnitude-polar 'magnitude)
(putprop 'polar 'angle-polar 'angle)

(defun operate (op obj)
  (let ((procedure (get (type obj) op)))
    (funcall procedure (contents obj))))

(defun real-part (obj)
  (operate 'real-part obj))
(defun imag-part (obj)
  (operate 'imag-part obj))
(defun magnitude (obj)
  (operate 'magnitude obj))
(defun angle (obj)
```

```
  (operate 'angle obj))
```

Message Passing

In the traditional style of programming the operator-type table was decomposed into rows, with each generic operator representing a row of the table. An alternative is to decompose the table into columns. Instead of using "intelligent operators" that dispatch on data types we work with "intelligent data objects" that dispatch on operator names.

Assume lexcial scoping. A data object, such as a rectangular number, is represented as a procedure that takes as input the required operation name and performs the operation needed.

Make-rectangular could be written as

```
(defun make-rectangular (x y)
 #'(lambda (message)
     (cond ((equal message 'real-part) x)
           ((equal message 'imag-part) y)
           ((equal message 'magnitude)
            (sqrt (plus (square x)
                        (square y))))
           ((equal message 'angle)
```

Data-directed programming

A weakness is that the generic interface procedures `real-part`, `imagpart` ... must know all the different complex number representations.

Two-dimensional table.

Represent the table explicitly.

```
(putprop 'rectangular 'real-part-rectangular
                      'real-part)
(putprop 'rectangular 'imag-part-rectangular
                      'imag-part)
(putprop 'rectangular 'magnitude-rectangular
                      'magnitude)
(putprop 'rectangular 'angle-rectangular
                      'angle)
(putprop 'polar 'real-part-polar 'real-part)
(putprop 'polar 'imag-part-polar 'imag-part)
(putprop 'polar 'magnitude-polar 'magnitude)
(putprop 'polar 'angle-polar 'angle)

(defun operate (op obj)
  (let ((procedure (get (type obj) op)))
    (funcall procedure (contents obj))))

(defun real-part (obj)
  (operate 'real-part obj))
(defun imag-part (obj)
  (operate 'imag-part obj))
(defun magnitude (obj)
  (operate 'magnitude obj))
(defun angle (obj)
```

```
            (atan y x)))))
```

The corresponding **operate** procedure becomes very simple.

```
(defun operate (op obj)
   (funcall obj op))
```

The name *message passing* comes from the image that a data object is an entity that receives the requested operation name as a message. This is the programming style used in **Object-oriented programming** which we will return to later.

The complex package can easily be expanded to a generic arithmetic that work on ordinary numbers, rational numbers and complex numbers.

Ordinary numbers

```
(defun +number (x y)
  (make-number (plus x y)))
(defun -number (x y)
  (make-number (difference x y)))
(defun *number (x y)
  (make-number (times x y)))
(defun /number (x y)
  (make-number (quotient x y)))

(defun make-number (n)
  (attach-type 'number n))
```

```
(putprop 'number '+number 'add)
(putprop 'number '-number 'sub)
(putprop 'number '*number 'mul)
(putprop 'number '/number 'div)
```

The actual generic operators are defined as follows:

```
(defun add (x y) (operate-2 'add x y))
(defun sub (x y) (operate-2 'sub x y))
(defun mul (x y) (operate-2 'mul x y))
(defun div (x y) (operate-2 'div x y))
```

The general procedure `operate-2` dispatches to the procedure that was installed in the table for the given type and operator.

```
(defun operate-2 (op arg1 arg2)
 (let ((t1 (type arg1)))
   (cond ((equal t1 (type arg2))
          (let ((proc (get t1 op)))
            (cond ((not (null proc))
                   (funcall proc (contents arg1)
                                 (contents arg2)))
                  (t (error "Undefined op")))))
         (t (error "Operands not of same type")))))
```

Interfacing the complex number package.

```
(defun make-complex (z)
  (attach-type 'complex z))

(defun +complex (z1 z2)
  (make-complex (+c z1 z2)))
```

```
(defun -complex (z1 z2)
  (make-complex (-c z1 z2)))

(defun *complex (z1 z2)
  (make-complex (*c z1 z2)))

(defun /complex (z1 z2)
  (make-complex (/c z1 z2)))

(putprop 'complex '+complex 'add)
(putprop 'complex '-complex 'sub)
(putprop 'complex '*complex 'mul)
(putprop 'complex '/complex 'div)
```

The operators `real-part`, `imag-part`, `magnitude` and `angle` are available only inside the complex number package. These can easily be exported so they can be applied directly to objects of type `complex`.

```
(defun real-part-complex (z)
  (make-number (real-part z)))

(defun imag-part-complex (z)
  (make-number (imag-part z)))

(defun magnitude-complex (z)
  (make-number (magnitude z)))

(defun angle-complex (z)
  (make-number (angle z)))

(putprop 'complex 'real-part-complex 'real-part)
(putprop 'complex 'imag-part-complex 'imag-part)
(putprop 'complex 'magnitude-complex 'magnitude)
(putprop 'complex 'angle-complex 'angle)
```

Flavors

Object-oriented package on top of Lisp.

Versions for different Lisp dialects.

Object: an instance of a flavor. Consists os a *local state* and some *behavior*.

Flavor: corresponds to the Simula, Smalltalk class concept.

The objects communicates by sending messages to each other that are taken care of by procedures called *methods*.

This paradigm permits implementation of generic algorithms. A set of messages (sometimes called a *protocol*) is defined that specifies what the external behavior must be if an object is to implement the protocol.

The protocol does however not define the internal implementation.

Supports multiple inheritance

Mixing flavors.

```
(load 'flavors)
```

```
(defflavor flavor-name
  (instance-variables)
  (component-flavors)
 options)

(defflavor moving-object
  (x-position y-position
   x-velocity y-velocity
   mass)
  ())

(defflavor ship
  (name
   (engine-power 100))
  (moving-object))

(defflavor cargo-freighter
  (capacity
   deadweight)
  ())

(defflavor tanker
  ()
  (ship cargo-freighter))
```

Options to automatically generate methods for accessing and retrieving instance variables, define default handlers for messages etc..

Instantiation.

```
(setq titanic (make-instance 'ship
                             'x-position 20
                             'y-position 45
                             'x-velocity 0
                             'y-velocity 0))
```

Method definitions.

```
(defmethod (flavor [messagetype] messagename)
   (arguments) body)

(defmethod (moving-object :speed) ()
  (sqrt (plus (square x-velocity)
              (square y-velocity))))

->(<- titanic ':speed)
0
```

If the option **gettable-instance-variables** is given then methods are automatically created for retrieving the values of the instance variables.

If the option **settable-instance-variables** is given then methods are created for setting the values of the instance variables.

```
->(<- titanic 'x-position)
20
->(<- titanic 'set-x-position 30)
30
```

Inheritance

The instance variables of an object is the union of all the variables of the components. If different components have the same name of an instance variable then all methods referring to this name will refer to the same variable.

When a flavor is defined a list of all the components are computed. This is done through a depth-first, left-right tree traversation with elimination of duplets. This list determines the order in which the system searches for methods.

Message types:

- primary
- before
- after
- wrapper

Vanilla

All flavors contain the component **Vanilla** as a default.

Vanilla has methods for the following messages;

- pretty-print
- describe
- which-operations
- describe
- etc.

7. SCHEME

- Introduction
- Assignment and local state
- Constraints example
- Streams
- Other Common Lisp features
- Lisp history

SCHEME

Gerald Sussman and Guy Steele MIT

"The structure and interpretation of computer programs" Abelson and Sussman

Lexical scoping.

Full funarg capabilities i.e. possible to have functional objects as returned values in a nice way.

No difference between the value and the function binding of a symbol.

```
->(define pi 3.14159)
pi
->(define radius 10)
radius
->(* pi (* radius radius))
314.159
->(car '(1 2 3))
1
->car
<primitive-procedure 123456>
->((car (list cdr car)) '(1 2 3))
(2 3)
```

```
->(define (sum-of-squares x y)
    (+ (square x) (square y)))
sum-of-squares
```

When a functional form is evaluated **all** the elements of the list are evaluated including the first.

```
(define (sum-loop term a next b)
   (if (> a b)
       0
       (+ (term a)
          (sum-loop term (next a) next b))))
```

i.e. no funcall or apply needed here.

```
(define (deriv f dx)
  (lambda (x)
     (/ (- (f (+ x dx)) (f x))
        dx)))

->((deriv cube 0.001) 5)
75.15
```

The first element of the list is evaluated.

Newtons method for finding the roots of a differentiable function.

```
(define (newton f guess)
  (if (good-enough? guess f)
      guess
      (newton f (improve guess f))))

(define (improve guess f)
  (- guess (/ (f guess)
              ((deriv f 0.001) guess))))

(define (good-enough? guess f)
  (< (abs (f guess)) 0.001))

->(newton (lambda (x) (- x (cos x))) 1)
0.7391
```

Assignment and Local State

Ex: Withdrawing money from a bank account

A procedure **withdraw** takes an argument *amount* to be withdrawn. If there is enough money in the account then **withdraw** should return the balance remaining after the withdrawal. If we begin with 100 dollars in the account, the following responses should be obtained.

```
->(withdraw 25)
75
->(withdraw 25)
50
->(withdraw 60)
Insufficient funds
->(withdraw 15)
35
```

Notice that the two expressions **(withdraw 25)** both executed in the same context, yield different values.

Withdraw 1.

```
(define balance 100)

(define (withdraw amount)
  (if (>= balance amount)
      (sequence (set! balance (- balance amount))
```

```
                balance)
      "Insufficient funds"))
```

Problem: The global variable balance is freely accessible to other procedures.

Withdraw 2.

```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (sequence (set! balance
                          (- balance amount))
                    balance)
          "Insufficient funds"))))
```

Let establishes an environment with a local variable `balance` bound to the initial value 100. Within this environment, we use lambda to create a procedure that takes amount as an argument and behaves correctly.

Withdraw 3. "withdrawal processors"

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds")))

(define w1 (make-withdraw 100))
```

```
(define w2 (make-withdraw 100))

->(w1 50)
50
->(w2 70)
30
```

w1 and w2 are completely independent objects, each with its own local state.

We can create objects that handle deposits as well as withdrawals and thus represent simple bank accounts.

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance
                        (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
     (cond ((eq? m 'withdraw) withdraw)
           ((eq? m 'deposit)  deposit)
           (else (error "Unknown request"))))
  dispatch)
```

Each call to **make-account** sets up an environment with a local state variable **balance**. Within this environment two procedures, **deposit** and **withdraw** are defined which access **balance**. An additional procedure **dispatch**, which takes a "message" as input and returns one of the local procedures is returned as the value that represents the account.

```
->(define acc (make-account 100))
acc
->((acc 'withdraw) 50)
50
->((acc 'deposit) 40)
90
```

Constraints

Programs are traditionally organized in terms of *one-directional* computations. They perform operations on pre-specified arguments to produce desired outputs. On the other hand, we often model systems in terms of relations among quantities.

```
U = R * I
```

Such an equation is not one-directional.

In this example we design a language that enables us to work in terms of relations themselves. The primitive elements of the language are *primitive constraints*. For example;

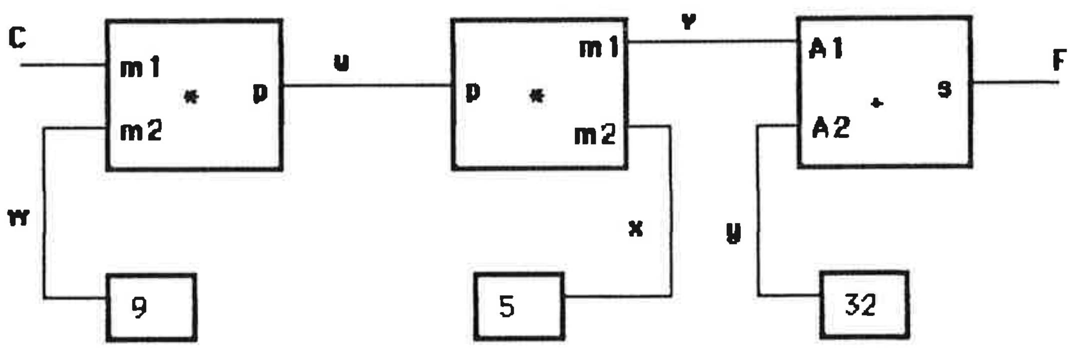
```
(adder a b c)

(multiplier x y z)

(constant <number> x)
```

The constraints are combined by constructing a *constraint network* in which constraints are joined via *connectors*. A connector is an object that ”holds” a value that may participate in one or more constraints.

Example: Conversion between Centigrade and Fahrenheit temperatures.

```
9 C = 5 (F - 32)
```

This can be thought of as a network.



The computation proceeds as follows: When a connector is given a value, it awakens all of its associated constraints to inform them that it has a value. Each awakened constraint box then polls its connectors to see if there is enough information to determine a value for a connector. If so, the box sets that connector which then awakens all of its associated constraints and so on.

```
(define C (make-connector))
(define F (make-connector))
(centigrade-fahrenheit-converter C F)

(define (centigrade-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)))
```

This procedure creates the internal connectors and links them as shown in the figure using the primitive constraint boxes.

To watch the network in action we can place probes on the connectors C and F using a **probe** procedure. Placing a probe on a connector will cause a message to be printed whenever the connector is given a value.

```
(probe "Centigrade temp" C)
(probe "Fahrenheit temp" F)
```

Next we set the value of C to 25.

```
->(set-value! C 25 'user)
Probe: Centigrade temp = 25
```

```
Probe: Fahrenheit temp = 77
done
```

The probe on C awakens and reports the value. C also propagates the value through the network which sets F to 77.

```
->(set-value! F 212 'user)
Error! Contradiction (77 212)

->(forget-value! C 'user)
Probe: Centigrade temp = ?
Probe: Fahrenheit temp = ?
done

->(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Centigrade temp = 100
done
```

Implemenation using procedural objects with local state.
The basic operations on the connectors are

```
(has-value? <connector>)
  tells whether the connector currently has a value.

(get-value? <connector>)
  returns the current value

(set-value! <connector> <value> <informant>)
  tells the connector that some informant is
  requesting it to set its value to a new value
```

```
(forget-value! <connector> <retractor>)
  tells the connector that some retractor is
  requesting it to forget its value

(connect <connector> <new-constraint>)
  tells the connector to participate in a new
  constraint
```

Connectors communicate with constraints using the procedures **inform-about-values** which tells the constraint that the connector has a value and **inform-about-no-value** that tells the constraint that the connetor has lost its value.

```scheme
(define (adder a1 a2 sum)
 (define (process-new-value)
   (cond ((and (has-value? a1) (has-value? a2))
          (set-value! sum
                      (+ (get-value a1)
                         (get-value a2))
                      me))
         ((and (has-value? a1) (has-value? sum))
          (set-value! a2
                      (- (get-value sum)
                         (get-value a1))
                      me))
         ((and (has-value? a2) (has-value? sum))
          (set-value! a1
                      (- (get-value sum)
                         (get-value a2))
                      me))))
 (define (process-forget-value)
    (forget-value! sum me)
    (forget-value! a1 me)
    (forget-value! a2 me)
    (process-new-value))
 (define (me request)
   (cond ((eq? request 'I-have-a-value)
          process-new-value)
         ((eq? request 'I-lost-my-value)
          process-forget-value)
         (else (error "Unknown request"))))
 (connect a1 me)
 (connect a2 me)
 (connect sum me)
 me)

(define (inform-about-value constraint)
  ((constraint 'I-have-a-value)))

(define (inform-about-no-value constraint)
  ((constraint 'I-lost-my-value)))
```

The multiplier is very similar.

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1)
                    (= (get-value m1) 0))
               (and (has-value? m2)
                    (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                       (* (get-value m1)
                          (get-value m2))
                       me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                       (/ (get-value product)
                          (get-value m1))
                       me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                       (/ (get-value product)
                          (get-value m2))
                       me))))
  (define (process-forget-value)
     (forget-value! product me)
     (forget-value! m1 me)
     (forget-value! m2 me)
     (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           process-new-value)
          ((eq? request 'I-lost-my-value)
           process-forget-value)
          (else (error "Unknown request"))))
  (connect m1 me)
  (connect m2 me)
```

```
  (connect product me)
  me)


(define (constant value connector)
  (define (me request)
     (error "Unknown request"))
  (connect connector me)
  (set-value! connector value me)
  me)


(define (probe name connector)
  (define (process-new-value)
    (print "Probe: ")
    (princ name)
    (princ " = ")
    (princ (get-value connector)))
  (define (process-forget-value)
    (print "Probe: ")
    (princ name)
    (princ " =  ? "))
  (define (me request)
    (cond ((eq? request 'I-have-a value)
           process-new-value)
          ((eq? request 'I-lost-my-value)
           process-forget-value)
          (else (error "Unknown request"))))
  (connect connector me)
  me)
```

Representing connectors: A connector is represented as a procedural object with local state variables `value`, the current value of the connector, `informant`, the object that set the value and `constraints`, a list of the constraints in which the connector participates.

```
(define (make-connector)
  (let ((value nil)
        (informant nil)
        (constraints nil))
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter
                              inform-about-value
                              constraints))
            ((not (= value newval))
             (error "Contradiction"
                    (list value newval)))))

    (define (forget-my-value retractor)
      (if (eq? retractor informant)
          (sequence
             (set! informant nil)
             (for-each-except retractor
                              inform-about-no-value
                              constraints))))

    (define (connect new-constraint)
      (if (not (memq new-constraint constraints))
          (set! constraints
                (cons new-constraint constraints)))
      (if (has-value? me)
          (inform-about-value new-constraint)))

    (define (me request)
      (cond ((eq? request 'has-value)
             (not (null? informant)))
            ((eq? request 'value) value)
            ((eq? request 'set-value!)
             set-my-value)
            ((eq? request 'forget)
```

```
                 forget-my-value)
                ((eq? request 'connect) connect)
                (else (error "Unknown operation"))))

    me))
```

```
(define (for-each-except except proc list)
  (define (loop items)
     (cond ((null? items) 'done)
           ((eq? (car items) except)
            (loop (cdr items)))
           (else (proc (car items))
                 (loop (cdr items)))))
```

Syntax interface

```
(define (has-value? connector)
  (connector 'has-value?))

(define (get-value connector)
  (connector 'value))

(define (forget-value! connector retractor)
  ((connector 'forget) retractor))

(define (set-value! connector newval informant)
  ((connector 'set-value!) new-value informant))

(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

And that's it.

Streams

Ex: A procedure that takes as argument a binary tree, all of whose leaves are integers and computes the sum of the squares of the odd ones.

```
(define (sum-odd-squares tree)
  (if (leaf-node? tree)
      (if (odd? tree)
          (square tree)
          0)
      (+ (sum-odd-squares (left-branch tree))
         (sum-odd-squares (right-branch tree)))))
```

Ex: A procedure that constructs a list of all the odd Fibonacci numbers `Fib(k)` where k is less than or equal to a given integer n.

```
(define (odd-fibs n)
  (define (next k)
    (if (> k n)
        '()
        (let ((f (fib k)))
          (if (odd? f)
              (cons f (next (+ 1 k)))
              (next (+ 1 k))))))
  (next 1))
```

A common pattern. The first program

- Enumerates the leaves of a tree;
- filters them, selecting the odd ones;
- squares each of the selected ones;
- accumulates the result by adding, starting with 0.

The second program

- Enumerates the integers from 1 to n;
- computes the Fibonacci number for each integer;
- filters them, selecting the odd ones;
- accumulates the result into a list, using cons, starting with the empty list.

ENUMERATE – FILTER – MAP – ACCUMULATE

View this as a signal or *stream* that flows through a cascade of stages.

The strems are defined abstractly as a constructor `cons-stream` and two selectors `head` and `tail`. They are related through as follows.

- For any objects a and b, if x is **(cons-stream a b)** then **(head x)** is a and **(tail x)** is b.

There also exists an objext called **the-empty-stream** and a predicate **empty-stream?**.

Example 1.

```
(define (enumerate-tree tree)
  (if (leaf-node? tree)
      (cons-stream tree the-empty-stream)
      (append-streams
        (enumerate-tree (left-branch tree))
        (enumerate-tree (right-branch tree)))))

(define (append-streams s1 s2)
 (if (empty-stream? s1)
     s2
     (cons-stream (head s1)
                  (append-streams (tail s1) s2))))

(define (filter-odd s)
  (cond ((empty-stream? s) the-empty-stream)
        ((odd? (head s))
         (cons-stream (head s)
                      (filter-odd (tail s))))
        (else (filter-odd (tail s)))))

(define (map-square s)
  (if (empty-stream? s)
      the-empty-stream
      (cons-stream (square (head s))
                   (map-square (tail s)))))
```

```
(define (accumulate-+ s)
  (if (empty-stream? s)
      0
      (+ (head s)
         (accumulate-+ (tail s)))))

(define (sum-odd-squares tree)
  (accumulate-+
     (map-square
        (filter-odd
           (enumerate-tree tree)))))

(define (odd-fibs n)
  (accumulate-cons
     (filter-odd
        (map-fib
            (enumerate-interval 1 n)))))
```

Higher order procedures for streams.

```
(define (accumulate combiner initial-value stream)
  (if (empty-stream? stream)
      initial-value
      (combiner (head stream)
                (accumulate combiner
                            initial-value
                            (tail stream)))))

(define (map proc stream)
  (if (empty-stream? stream)
      the-empty-stream
      (cons-stream (proc (head stream))
```

```
                    (map proc (tail stream)))))

(define (filter pred stream)
  (cond ((empty-stream? stream) the-empty-stream)
        ((pred (head stream))
         (cons-stream (head stream)
                      (filter pred (tail stream))))
        (else (filter pred (tail stream)))))
```

Inefficient. Example: Compute the second prime in the interval 10000 to 1000000.

```
(head
  (tail
    (filter prime?
            (enumerate-interval 10000 1000000))))
```

The problem is solved by arranging for **cons-stream** to construct a stream only partially and to pass the partial construction to the program that consumes the stream. If the consumer attempts to a access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more of itself to enable the consumer to access the required part, thus preserving the illusion that the entire stream exists. This is called *lazy evaluation.*

This is done by arranging for the tail of a stream to be evaluated, not when the the stream is constructed by **cons-stream** but rather when the tail is accessed by the **tail** procedure.

The special form **(delay expression)** is used which does not evaluate th expression but rather returns a so-called *delayed object*, which we can think of as a "promise" to evaluate the expression at some future time. To evaluate this delayed object the operator **force** is used.

```
(cons-stream a b) == (cons a (delay b))

(define (head stream) (car stream))
(define (tail stream) (force (cdr stream)))
```

Example: Prime computation.

```
(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                   (enumerate-interval (+ low 1)
                                       high))))

(head
   (tail
     (filter prime?
             (enumerate-interval 10000 1000000))))
```

Infinitely long streams

```
(define (integers-starting-from n)
  (cons-stream n (integer-starting-from (+ n 1))))

(define integers (integers-starting-from 1))

(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
```

The *sieve of Eratosthenes* method for primes.

We start with the integers starting with 2, which is the first prime. To get the rest of the primes, we start by filtering the multiples of 2 from the rest of the integers. This leaves a stream beginning with 3, which is the next prime. Now we filter the multiples of 3 from the rest of the stream and so on.

In other words, to sieve a stream S, we form a stream whose head is the head of S and whose tail is obtained by filtering all multiples of the head of S out of the tail of S and sieving the result.

```
(define (sieve stream)
  (cons-stream
    (head stream)
```

```
    (sieve (filter
             (lambda (x) (not
                           (divisible?
                              x
                              (head stream))))
             (tail stream)))))
```

```
(define primes (sieve (integer-starting-from 2)))

(define (divisible? a b)
 (= (remainder b a) 0))


(define (print-stream stream)
  (print (head stream))
  (print-stream (tail stream)))

->(print-stream primes)
2
3
5
7
11
.
.
```

Infinite streams can also be defined implicitly.

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
  (cond ((empty-stream? s1) s2)
        ((empty-stream? s2) s1)
        (else (cons-stream
                (+ (head s1) (head s2))
```

```
                    (add-streams (tail s1) (tail s2))))))
```

```
(define integers
  (cons-stream 1
    (add-streams ones integers)))
```

Streams can be used to model signal processing systems, representing the values of a signal at successive time intervals as consecutive elements of a stream. As an example we can implement as integrator or summer that for an input stream, an initial value and a small increment dt accumulates the sum.

```
(define (integral integrand initial-value dt)
  (define int
     (cons-stream initial-value
                  (add-streams
                    (scale-stream dt integrand)
                    int)))
  int)
```

Pure functional programming. Local state without assignment.

Other Common Lisp features

Packages, different name-spaces for symbols, exporting and importing.

Generic operations on sequences (lists and vectors). Ex: length, reverse ...

Complex numbers and rational numbers as standard data types.

Hash tables as standard data types.

Short History

1960 John McCarty Lisp 1.5

Developed into two principal dialects.

Interlisp - BBN, Xerox, Stanford, West Coast

MacLisp - MIT, (Franz Lisp Berkeley).

Standardization attempts

PSL- Portable Standard Lisp

Common Lisp

1975 Lispmachines

MIT – Symbolics, LMI, ZetaLisp

Xerox – Interlisp.

EXERCISE 1.

Problem 1: Write sequences of car and cadr that will pick the symbol PEAR out of the following expressions:

```
(apple orange pear grapefruit)

((apple orange) (pear grapefruit))

(((apple) (orange) (pear) (grapefruit)))

((((apple))) ((orange)) (pear) grapefruit)
```

Problem 2: Define rotate-left, a function that takes a list and returns a new list in which the former first element becomes the last.

```
->(rotate-left '(a b c))
(b c a)
```

Problem 3: A palindrome is a list that has the same sequence of elements when read from right to left as when read from left to right. Define palindrome such that it takes a list as its argument and returns a palindrome that is twice as long.

```
->(palindrome '(a b c d))
(a b c d d c b a)
```

Problem 4: Represent a point in the plane as a two-element list. Write a function that takes two such lists and returns the Euclidian distance between the points.

Problem 5: One of the more complicated recursions occur in the so called Ackermann's function.

```
(defun Ack (x y)
  (cond ((equal y 0) 0)
        ((equal x 0) (* 2 y))
        ((equal y 1) 2)
        (t (Ack (- x 1)
                (Ack x
                     (- y 1))))))
```

Give a mathematical definition for each of the following functions.

```
(defun f (n) (Ack 0 n))

(defun g (n) (Ack 1 n))

(defun h (n) (Ack 2 n))
```

Problem 6: Express (abs x), (min a b) and (max a b) in terms of cond.

Problem 7: Write a recursive function remove that removes all occurences of an element from a) the top-level of a list b) all levels of a list.

Problem 8: Define our-reverse, a tail-recursive version of reverse.

Problem 9: Define squash, a procedure that takes an expression as its argument and returns a nonnested list of all atoms found in the expression.

```
->(squash '(a (a (a b)) c (c d)))
(a a a b c c d)
```

Problem 10: Binary trees can be used to represent arithmetic expressions, as ffor example:

```
(* (+ a b) (- c (/ d e)))
```

Part of the work of a compiler is to translate such an arithmetic expression into the machine language of some computer. Suppose that the target machine has a set of sequentially numbered registers that can hold temporary results.
Further suppose that the target machine has a MOVE instruction for getting values into registers and ADD, SUB, MUL, and DIV for arithmetically combining values in two registers. The example could be translated as follows:

```
((move 1 a)
 (move 2 b)
 (add 1 2)
 (move 2 c)
 (move 3 d)
 (move 4 e)
 (div 3 4)
 (sub 2 3)
 (mul 1 2))
```

The result of a calculation is left in the first register. Define compile-arithmetic, a procedure that performs this translation.

EXERCISE 2.

Problem 1: Solve problem 7 of Exercise 1 with the use of mapcar.

Problem 2: Write your own version of mapcar using list recursion. You can assume that mapcar only takes two arguments.

Problem 3: Suppose a matrix is represented as a list of lists. For example ((a b) (c d)) would represent the 2x2 matrix whose first row contains a b and whose second row contains c d. Write a function that takes a matrix as input and outputs its transpose.

Problem 4: A useful function that combines flow of control and function mapping is called some. This is a function of two arguments, which should evaluate to a function and a list. It applies the function to successive elements of the list until the function returns non-nil. Then it returns the elements of the list from that point on. It returns nil otherwise. For example (some 'numberp '(a b 2 c d)) should return (2 c d).

The function every is like some except that it stops as soon as one of the function applications returns nil. Every then returns nil as its value. If all the application return non-nil, every returns t.

Write the function some and every.

Problem 5: Write a version of cons called mcons that takes any number of arguments, all of which are evaluated. The value of the next-to-last argument should be consed onto the last; the value before should be consed onto the resulting value, and so on. For example, (mcons 'a 'b 'c '(d e)) should return (a b c d e).

Problem 6: The functions remove-if and remove-if-not are used to do filtered accumulations. These functions takes two arguments. The first should evaluate to a function and the second to a list. Remove-if returns a list of all the elements in the list for which the functions evalutes to nil and remove-if-not returns a list of all the elements for which the functions evaluates to non-nil.

```
->(remove-if-not 'fruitp '(apple corn milk pear))
(apple pear)

->(remove-if 'fruitp '(apple corn milk pear))
(corn milk)
```

Write the functions remove-if and remove-if-not.

Problem 7: Sum-loop is a special case of the more general function accumulate which combines a collection of terms, using the general accumulation function:

```
(accumulate combiner null-value term a next b)
```

Accumulate takes the same arguments as sum-loop together with a combiner function of two arguments that specifies how the current term is to be combined with the accumulation of the preceding terms and a null-value the specifies what initial value to use when the terms run out. Write accumulate (both in recursive and tail-recursive forms) and show how sum-loop is defined in terms of accumulate.

EXERCISE 3.

Problem 1: In addition to IF, Common Lisp has WHEN and UNLESS, defined as follows:

```
(when <test> <forms>) == (cond (<test> <forms>))

(unlesss <test> <forms>) ==(cond ((not <test>) <forms>))
```

Define the macros when and unless.

Problem 2: Not all Lisp systems have the backquote mechanism. Define backquote such that it has the effect of backquote and allows for the appropriate handling of expressions with COMMA and COMMA-AT, as in the following case.

```
->(backquote (a b (list 'c 'd) (comma (list 'e 'f)) (comma-at (list 'g 'h))))
(a b (list 'c 'd) (e f) g h)
```

Problem 3: Suppose Let did not exist. Define your own let as macro using lambda.

Problem 4: The pure Lisp system makes it difficult to keep track of which top-level symbols you have given value with setq. Define a macro assign that behaves like setq but also store the symbol name and the value on a global association list called *my-variables*. Define procedures that return all the assigned symbols with and with their values.

Problem 5: Define a macro while that can be called with expressions like

```
(while <test> <forms>)
```

As long as the expression test evaluates to true, while repeatedly evaluates the forms.

Projects

1. Extend the symbolic differentiation example. Include rules for trigonometric and exponential expressions. Allow for input on ordinary infix notation with the usual precedence rules. Return the result on a nice "Macsyma" form. Allow for substitutions in expressions and evaluation of expressions.

2. Extend the generic arithmetic package from seminar 6. Allow other data types such as polynomials, matrices etc.

3. Build an implementation of constraints using Flavors. Find a more appropriate example than the centigrade to fahrenheit converter. Ex. Electric network with resistors, capacitors, solenoids etc..

4. Try to implement a general object-oriented package like Flavors in Scheme with functional objects and message passing. The system should handle inheritance correctly.

5. Combine the streams based Prolog interpreter from Abelson - Sussman and the streams based forward chaining expert system shell from Winston - Horn into an expert system shell with both forward and backward chaining.

6. Write Adventure type game using objects that represent the entities in the world. Do it with Flavors or in Scheme.

7. Own ideas.

Förslag till Lisp-uppgifter

Bengt Mårtensson, 1986-01-05

Jag har en del förslag till mindre programmeringsuppgifter i Lisp. Dessa är nerslängda i all hast, men här kommer dom. Jag bryr mig inte om att nämna datalogiska standarduppgifter så som symbolisk derivering etc. Till stor del kommer detta att vara TEX-relaterade "utilities". Jag tycker dessa är "verklighetsanknutna" uppgifter som kan användas till nånting.

Index-program för TEX

Bakgrund: Både jag själv och "adaptiva-bok-gänget" håller på att skriva TEX så det dundrar om det. Vi behöver bra hjälpmedel för att hantera index. Detta ska klara av:
* Vanliga referenser
* "Primary references", dvs här ska sidnumret sättas i fetstil
* "span"-referenser, dvs ett sidintervall av typen 23–29.
* referenser till fotnötter
* Korsreferenser ("See ..." och "See also...")
* Speciall sorterings"tag" för TEX-macron. T ex ska man kunna få makroanropet `!MyFavoriteLieGroup` insorterat som $SO(n)$.
* Hierarkisk referenslista.

Vid första passet genererar TEX en textfil med osorterade referenser. Lisp-programmet sorterar denna och gör en ny TEX-fil av den, som sen TEX-as i sin tur. Se TUG-boat nr 1/1. Syntaxen är dock knäpp, och jag har en bättre i huvudet. TUG-boaten innehåller också ett interlisp-programm som påstås fungera. Uppgiften består i att skriva Lisp-programmet. TEX-macrona kan jag och/eller Leif göra (såvida ingen annan är intressered??). Inga TEX-kunskaper är nödvändiga.

Konvertering av DOC-filer till TEX-format

Bakgrund: Man stöter allt som oftast på en massa DOC-filer med information man är intresserad av. Dessa är väldigt ofta i ett format som är olämpligt, och man kanske skulle vilja ha det utskivet lite snyggare. Ofta har det körts genom någon form av ordbehandlingsprogram, och innehåller indenteringar, tabeller, rubriker etc. Dock innehåller de inte matematik.

Uppgiften består i att skriva lisp-funktioner som generarar TEX-kod från en DOC-fil. Rubriker, tabeller, displayer etc ska detekteras och hanteras förnuftigt. Tecken som betyder nåt speciellt för TEX ska "escapas" (vad säger man på svenska?), eller så kan man ändra på catcoden. Eventuellt kan programmet interagera med användaren, t ex fråga "Ska jag behandla detta som en tabell?". Observera att man kan göra denna uppgift mer eller mindre fullständigt, och att man nog ska se utmatningen som en första iteration, i varje fall om man har höga krav. Ganska goda TEX-kunskaper nödvändigt.

Report till TEX-konverterare

Rubriken talar för sig själv. F ö gäller kommentarerna ovan här också.

NROFF/TROFF till TEX-konverterare

Vill bara kommentera att det finns en del intressant text i nroff/troff-format på EUNICE-arean. T ex UNIX- och Franz-manualer.

Behandling av MACSYMA-resultat

Motivering: att kunna läsa MACSYMA output från andra program, att kunna göra nåt vettigt med det (t ex plotta grejsmojs; tyvärr saknar vi MACSYMAS plottrutiner), att kunna plocka in det direkt i textfiler, t ex i TEX (där kom det!).

MACSYMA är skrivet i Franz Lisp, och man kan komma ner i Lisp-en med ctrl-Z. Därifrån kan man direkt komma åt den interna representationen av uttrycken. Jag har själv skrivit två funktioner, `matpr` och `polypr` som skriver ut matriser och polynom på en textfil. Notera att MACSYMA kan fås att automatiskt generara FORTRAN-kod. Kanske kan man generera en plot-fil, antingen i T4010-format, CANON-format eller POSTSCRIPT. Eller varför inte Turtlegraphics i LOGO? Referens: Macsyma-manualen, speciellt kapitel 10.

"Kompilator" för enkelt språk för beskrivning av block-diagram

Uppgiftern består i att definiera ett enkelt språk för beskrivning av blockdiagram, och att skriva LISP-funktioner som ur en beskrivning i detta språk genererar antingen T4010-, TEX- POSTSCRIPT-, eller LOGO-kod. Detta skulle fungera ungeför som `pic` eller `ideal` i UNIX, se Kernighan-Pike sid 313. Vad det gäller TEX, kolla upp TUG-boat nr 2(1985) sid 83 och nr 3(1985). (Det senare numret innehåller också makron för listhantering i TEX, och makron som direkt sätter lösningen till "Towers of Hanoi"-problemet. Häftigt.) Notera också att LATEX innehåller möjlighet att dra sneda linjer, vilket är implementerat genom att man har en särskild font för detta.

Digitalteknik

Inom digitaltekniken (eller mera allmänt vad som behandlar finita automata) finns det ett antal algoritmer som lämpar sig väl för listprogrammering. Jag tänker på t ex tillståndsminimering, finna "billigaste" implementeringen av en kombinatorisk funktion med Quine - McCluskeys metod, bestämmandet av den uppnåeliga delmängden av tillståndsrummet från ett givet begynnelsetillstånd, SP-kodning av tillstånden, mm mm. Själv skrev jag en gång ett lisp-program för tillståndsminimering. Referens: Johannesson, Digitalteknik. Kolla gärna också nån mera teoretisk bok, t ex Manna eller Lewis - Papadimitriou (jag har dessa på mitt rum).

ANSWERS 1

Problem 1. caddr, caadr, caaddar, caaddr

Problem 2

```
(defun rotate-left (l)
  (append (cdr l) (list (car l))))
```

Problem 3

```
(defun palindrome (l)
  (append l (reverse l)))
```

Problem 4

```
(defun euclid-distance (p1 p2)
  (sqrt (plus (square (diff (xcoord p1) (xcoord p2)))
              (square (diff (ycoord p1) (ycoord p2))))))

(defun square (x) (expt x 2))

(defun xcoord (p) (car p))
(defun ycoord (p) (cadr p))
```

Problem 5

```
(defun ack (x y)
  (cond ((equal y 0) 0)
        ((equal x 0) (* 2 y))
        ((equal y 1) 2)
        (t (ack (- x 1)
                (ack x
                     (- y 1))))))
```

```
f(n) = 2n
g(n) = 0     n = 1
       2^n  else
h(n) = 0     n = 1
       2^h(n-1) else
```

Problem 6

```
(abs x) == (cond ((greaterp x 0) x)
                 (t (minus x)))
(min a b) == (cond ((greaterp a b) b)
                   (t a))
(max a b) == (cond ((greaterp a b) a)
                   (t b))
```

Problem 7

```
(defun remove-top (e l)
  (cond ((null l) nil)
        ((equal (car l) e)
         (remove-top e (cdr l)))
        (t (cons (car l)
                 (remove-top e (cdr l))))))

(defun remove-all (e l)
  (cond ((null l) nil)
        ((equal (car l) e)
         (remove-all e (cdr l)))
        ((listp (car l))
         (cons (remove-all e (car l))
               (remove-all e (cdr l))))
        (t (cons (car l)
                 (remove-all e (cdr l))))))
```

Problem 8

```
(defun our-reverse (list)
  (reverse-iter nil list))


(defun reverse-iter (result input)
  (cond ((null input) result)
        (t (reverse-iter
             (cons (car input) result)
             (cdr input)))))
```

Problem 9

```
(defun squash (l)
  (cond ((null l) nil)
        ((atom l) (list l))
        (t (append (squash (car l))
                   (squash (cdr l))))))
```

Problem 10

```
(defun compile-arithmetic (s)
  (compile-arithmetic-aux 1 s))

(defun compile-arithmetic-aux (register expr)
  (cond ((atom expr) (list (list 'move register expr)))
        (t (append
             (compile-arithmetic-aux register (cadr expr))
             (compile-arithmetic-aux (+ register 1) (caddr expr))
             (list (list (opcode (car expr)) register (+ register 1)))))))

(defun opcode (op)
  (cond ((equal op '+) 'add)
        ((equal op '-) 'sub)
        ((equal op '*) 'mul)
        ((equal op '/) 'div)
        (t 'err)))
```

ANSWERS 2

Problem 1

```
(defun remove-top (element l)
  (apply 'append
         (mapcar '(lambda (x)
                    (cond ((equal x element) nil)
                          (t (list x))))
                 l)))


(defun remove-all (element l)
  (apply 'append
         (mapcar '(lambda (x)
                    (cond ((equal x element) nil)
                          ((listp x) (list (remove-all element x)))
                          (t (list x))))
                 l)))
```

Problem 2

```
(defun my-mapcar (func list)
  (cond ((null list) nil)
        (t (cons (funcall func (car list))
                 (my-mapcar func (cdr list))))))
```

Problem 3

```
(defun transpose (matrix)
  (apply 'mapcar (cons 'list matrix)))
```

Problem 4

```
(defun some (func list)
  (cond ((null list) nil)
        ((funcall func (car list)) list)
        (t (some func (cdr list)))))


(defun every (func list)
  (cond ((null list) t)
        ((funcall func (car list))
         (every func (cdr list)))
        (t nil)))
```

Problem 5

```
(defun mcons (&rest arguments)
  (cond ((equal (length arguments) 1) (car arguments))
        (t (cons
             (car arguments)
             (apply 'mcons (cdr arguments))))))
```

Problem 6

```
(defun remove-if (func list)
  (cond ((null list) nil)
        ((funcall func (car list))
         (remove-if func (cdr list)))
        (t (cons (car list)
                 (remove-if func (cdr list))))))
```

or

```
(defun remove-if-2 (func list)
  (apply 'append
         (mapcar '(lambda (x)
                      (cond ((funcall func x) nil)
                            (t (list x))))
                  list)))
```

```
(defun remove-if-not (func list)
  (cond ((null list) nil)
        ((funcall func (car list))
         (cons (car list)
               (remove-if-not func (cdr list))))
        (t (remove-if-not func (cdr list)))))
```

or

```
(defun remove-if-not-2 (func list)
  (apply 'append
         (mapcar '(lambda (x)
                      (cond ((funcall func x) (list x))
                            (t nil)))
                  list)))
```

Problem 7

```
(defun accumulate (combiner null-value term a next b)
  (cond ((greaterp a b) null-value)
        (t (funcall combiner (funcall term a)
                             (accumulate combiner
                                         null-value
                                         term
                                         (funcall next a)
                                         next
                                         b)))))
```

ANSWERS 3

Problem 1

```
(defmacro when (test &rest result)
  `(cond (,test ,@result)))

(defmacro unless (test &rest result)
  `(cond ((not ,test) ,@result)))
```

Problem 2

```
(defmacro backquote (s)
  (list 'backquote1 (list 'quote s)))

(defun backquote1 (s)
  (cond ((or (null s) (atom s)) s)
        ((equal (car s) 'comma) (eval (cadr s)))
        ((and (not (atom (car s))) (equal (caar s) 'comma-at))
         (append (eval (cadar s)) (backquote1 (cdr s))))
        (t (cons (backquote1 (car s)) (backquote1 (cdr s))))))
```

Problem 3

```
(defmacro our-let (argument-list &rest body)
  `((lambda ,(mapcar 'car argument-list) ,@body)
    ,@(mapcar 'cadr argument-list)))
```

Problem 4. Assign should be used instead of setq on top-level. An association list *global-variable-list* is used to keep track of the variables. (variables) returns a list of the defined variables.

```
(defmacro assign (variable value)
  (cond ((and (boundp '*global-variable-list*)
              (assoc variable *global-variable-list*))
         `(progn (assign-new-value (quote ,variable) ,value)
                 (setq ,variable  ,value)))
        ((boundp '*global-variable-list*)
         `(progn (assign-new-variable (quote ,variable) ,value)
                 (setq ,variable ,value)))
        (t `(progn (setq *global-variable-list*
                   (quote ,(list (list  variable value))))
             (setq ,variable  ,value)))))

(defun assign-new-value (var val)
  (let ((element (assoc var *global-variable-list*)))
    (setq *global-variable-list* (cons (list var val)
          (remove element *global-variable-list*)))))
```

```
(defun assign-new-variable (var val)
  (setq *global-variable-list* (cons (list var val)
                                     *global-variable-list*)))

(defun variables ()
  (mapcar '(lambda (x)
             (car x)) *global-variable-list*))
```

Problem 5. This version always returns nil.

```
(defmacro while (test &rest body)
  `(prog () loop (cond ((not ,test) (return nil)))
              ,@body
              (go loop)))
```