# Artificial Intelligence

## *INTRODUCTION*

# WHAT IS ARTIFICIAL INTELLIGENCE (AI)?

- Many definitions of Intelligence
- Aspects to consider:
  - Ability to solve problems
  - Ability to use knowledge - reasoning
  - Ability to Learn
  - Etc.
- Measure of intelligence: IQ (intelligence tests)…
- Does playing chess denote intelligence?
  - In 1980 apparently the answer was consensual…
  - Today there are machines that beat the World Champion!

# JOB DEFINITION

- There is no globally accepted definition.
- Elaine Rich: "Study of how to make computers perform tasks that at the moment people are better at."

- At one point people were better than machines at doing arithmetic operations!

# BIRTH OF AI

- The designation "Artificial Intelligence" was invented in 1956
  - Lecture at Darthmouth College, USA.
  - By:
    - John McCarthy, ... LISP (1959)
    - Marvin Minsky, ... Percetron (et. Seymour Papert) (1969)
    - Allen Newell, ... Logic Theorist (1956)
    - Arthur Samuel, ... Game Theory, Checkers (1963)
    - Herbert Simon ... Limited Rationality (Nobel Prize)

# LISP

## Introduction

# FUNCTIONAL PARADIGM

- Functions are "first class entities" i.e. they are treated as values and can be arguments to another function, i.e. define "higher order" functions, or they can be the result of invoking a function.
  - This is rare in imperative or object-oriented languages, and in most cases higher-order functions are not supported.
  - They can be used in any operation and can be created in run-time.
- Features of pure functional programming:
  - Functions have one parameter and one result.
    - There are no side effects; it implies referential transparency.
      - Data immutability: you do not modify data structures; if necessary you create new structures that are modified copies of the previous ones.
    - No value assignment to variables
    - No syntactic or semantic anomalies
  - Main control structure: recursion
    - taking advantage of the possibility of defining higher-order functions
- Functional programming follows from the lambda calculation.
- More information at: https://thesocietea.org/2016/12/core-functional-programming-concepts/

# LAMBDA CALCULATION

- A formal system based on mathematical logic used to express computation based on function abstraction and its application using variable bindings and substitutions.

- -calculus
  - Represents a universal computing model (*Turing-complete*)
- $x.x^2 +1$ is a lambda abstraction for the function: $f(x)=x^2 +1$
- Application: f(3)=10 ... $(x.x^2 +1)$ 3 10
- In lambda calculus functions can be used as input values of other functions. Example:

  $((x.x^2 +1) ((x.x-4) 7))$ 10
- Lambda functions can be labeled (with a name):

  f := $(x.x^2 +1)$ g:= $(x.x-4)$

  And now you can invoke (f (g 7)) 10

Alonzo Church

# BASIC CONCEPTS

- It is convenient to review the following concepts:
  - Compilation vs. Interpretation
    - Features
    - Advantages and disadvantages
  - Memory Management
    - Heap vs. Stack
    - Features
    - Advantages and disadvantages
    - Methods
  - Data Types
    - Semantics
    - Static vs. Dynamic Types

# COMPILATION VS. INTERPRETATION

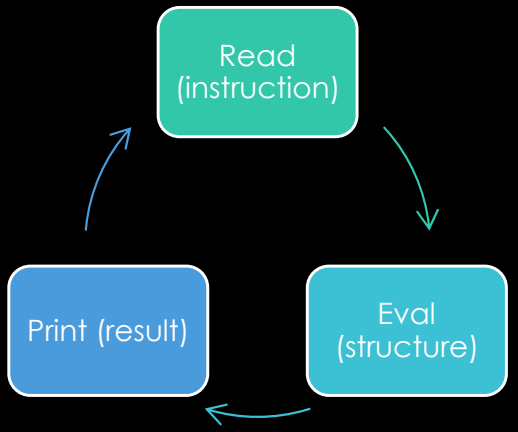
- **Compilation**: conversion to machine language before execution
  - Advantages:
    - Speed, but platform-dependent
    - It allows the detection of errors in the compilation,
    - Slow Debug: Testing new versions forces to recompile the program
- **Interpretation**: REPL
  - Advantages:
    - Platform Independence
    - Reflection (ability for the program to change itself: Assembler, Lisp, etc.)
      - Useful for testing: creating and instantiating entities during execution.
      - Essential in metaprogramming: treat programs as data.
        - https://en.wikipedia.org/wiki/Metaprogramming
    - Dynamic data types
      - Late binding / Duck Typing: type set only at runtime (not compile time)
      - https://en.wikipedia.org/wiki/Type_system#Dynamic_type-checking_and_runtime_type_information
    - Smaller program size
- Beyond this:
  - Just-in-time compilation technologies allow the gap between compiled and interpreted languages to be narrowing.
  - The "bytecode" languages allow you to compile to a virtual machine that then runs the program efficiently on different platforms, with advantages of both paradigms.

# CHARACTERISTICS OF A REFLECTIVE INTERPRETIVE LANGUAGE

- Interpreted: based on a REPL
  - Compiled in parts
  - It allows fast incremental testing, taking advantage of the interpreter.

Read-Eval-Print Cycle

Read-Eval-Print Loop (REPL)




- Reflexive: treats data as if it were programs
  - Direct access to the evaluator
  - It allows you to build code at runtime and therefore create data structures that are evaluated.

# MEMORY MANAGEMENT: STACK VS. HEAP

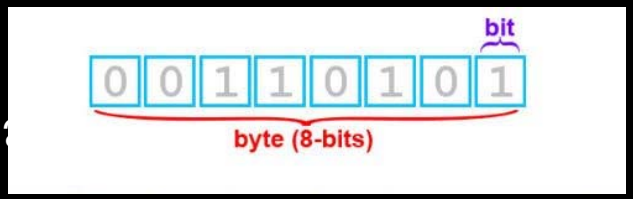
- **Stack:**
  - It follows a LIFO discipline, created in RAM for each thread, where the function invocation instances (frames) including code and variables are stored.
    - It serves for local, fixed-length, and limited-lived variables.
    - Fast, can use CPU cache, but limited (subject to overflow)
      - *Possible problem with recursion*
- **Heap:**
  - It also exists in RAM, but allows dynamic memory allocation, without CPU intervention.
    - It is for global variables of variable size and indefinite life.
    - Slow, but the memory limitation is the total memory of the machine
    - Interact with the heap through pointers
    - Management can be garbage collection based (as in Java or LISP) or manual (as in C or C++).
      - *If it is manual it can lead to memory leaks.*

# DATA TYPES

- The data type system of a language defines the semantics and behavior rules of the program.

whole: 49 ?

character: 1 ?




| | | | | | |
|---|---|---|---|---|---|
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |

- Advantages:
  - Abstraction and Modularity (interoperability)
  - Documentation (clarifies the programmer's intent)
  - It allows you to maximize, when compiling: optimization and security.
- Data Verification:
  - Static (in the source code text)
    - Can detect errors during compilation
  - Dynamics (during program execution)
    - Can detect errors during execution (e.g. CommonLisp)

# VARIABLES

- Main characteristics of a variable:
  - Type
  - Lifetime of a variable
    - From creation to disappearance: see stack vs. heap analysis.
  - Scope of a variable (scope)
    - Lexicon - defined by parsing the source code text: a variable can be referenced in blocks internal to the one in which it was defined.
      - This concept is associated with that of ***closure***
    - Dynamic - defined by the way the invocation of functions is chained together throughout the program execution.

# LISP KEY FEATURES

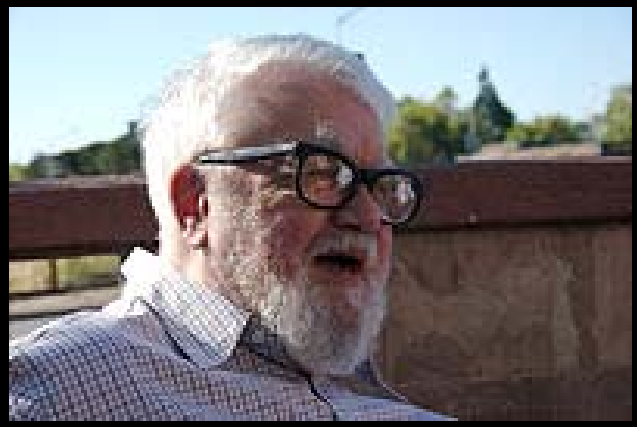
- Language
  - Functional
  - Performed
  - Reflexive
- Dynamic data types (late binding)
- Scope of variables defined lexically
- Pervasive dynamic data structures, taking extensive advantage of the *heap* but in which *pointers* are handled implicitly (transparently) through the data type: "List".

# GENESIS

- LISP = LISt Processing 1958
- Artificial Intelligence

Timeline of Lisp dialects(edit)

| | 1955 | 1960 | 1965 | 1970 | 1975 | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lisp 1.5 | Lisp 1.5 | | | | | | | | | | | | |
| Maclisp | | | Maclisp | | | | | | | | | | |
| Interlisp | | | | Interlisp | | | | | | | | | |
| ZetaLisp | | | | | Lisp Machine Lisp | | | | | | | | |
| Scheme | | | | | Scheme | | | | | | | | |
| NIL | | | | | NIL | | | | | | | | |
| Common Lisp | | | | | | Common Lisp | | | | | | | |
| T | | | | | | | T | | | | | | |
| Emacs Lisp | | | | | | | Emacs Lisp | | | | | | |
| AutoLISP | | | | | | | AutoLISP | | | | | | |
| ISLISP | | | | | | | ISLISP | | | | | | |
| EuLisp | | | | | | | EuLisp | | | | | | |
| Racket | | | | | | | | | Racket | | | | |
| Arc | | | | | | | | | | Arc | | | |
| Clojure | | | | | | | | | | | Clojure | | |
| LFE | | | | | | | | | | | LFE | | |
| Hy | | | | | | | | | | | | Hy | |



(John McCarthy)

Main dialects:

Stanford LISP, MacLisp, InterLisp, Franz Lisp, Zeta Lisp, Common Lisp (1984), **ANSI Common Lisp** (1999), ...

Development Environments: Lispworks, ...

# LISP: A FUNCTIONAL LANGUAGE

- The central element of LISP is **the Function**.
  - A program in LISP is a function: it returns a value rather than producing side effects, such as changing objects or assigning values to variables.
- Programming Style: bottom-up and top-down
  - Divide and conquer: top-down
    - Advantage: Reduces complexity
  - Increase the vocabulary of the language: "bottom-up
    - Advantages: makes programs easier to read, promotes code reuse, and allows you to discover useful patterns for the application domain.
- The language and the program evolve together.
- It is important to create abstractions adjusted to the application domain.

# PURE" LISP

- "Pure" LISP is a core language based exclusively on the lambda calculus, to which the various dialects later added many other ancillary features, some of which detracted from the functional paradigm.
- Key features:
  - Variables can change type during execution
  - Absence of an allocation transaction
  - No sequencing
  - Absence of iterative control structures (cycles)
  - Absence of the "pointer" concept for manipulating dynamic data structures
  - No need to manage the heap memory.
  - Interpreted (compiled in parts)
  - Programs and data have the same structure
  - Direct access to the evaluator which means that it is possible to make programs that make programs

# BASIC ASPECTS OF THE EVALUATOR: PROGRAMS VS. DATA

- Invoking Functions
  - You use lists with the following syntax:
    prefixed and bracketed notation
    (+ 2 3) => 5
- Plica
  - serves to avoid evaluating the expression on the right, treating it as data and not as a program.
    - 'a => a
    - '(a b) => (a b)
- Non-case-sensitive
  - You can write function and variable names case-insensitive.

# ELEMENTARY DATA TYPES

- Atoms
  - Symbols
  - Numbers
    - Integers
      - Fixnums
      - Bignums
    - Real
    - Etc.
  - Booleans
  - Characters
  - Strings
  - Other

- Lists
  - Dynamic data structures, used for data and programs

# LITERALS OF: NUMBERS, BOOLEANS, CHARACTERS, AND STRINGS

Examples of literals

Integers: -1 1 2
- Fixnum: [most-negative-fixnum, most-positive-fixnum]
- Bignum: 64646765756756756752343

Real: 1.2 5.3 4.1e7

Boolean: T NIL

Characters: #\a #\b

Strings: "I am a string"

# SYMBOLS

- Allow you to represent variables and functions
- Structure of a symbol (slots)
    - Name, function, value, property list, package

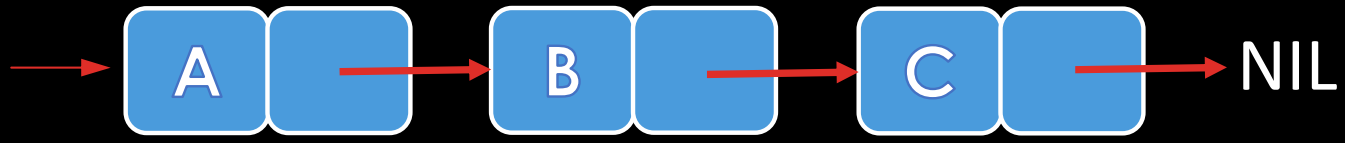
| factorial |
| --- |
| Name: "factorial |
| Function: (lambda(...)... |
| Value: nil |
| PList: nil |
| Package: default |

- The symbol table

| Factorial | A | B | Car | Cdr ... |
| --- | --- | --- | --- | --- |

# LISTS

- Lists: Dynamic Data Structures
  - Cell-based *cons*
    - https://en.wikipedia.org/wiki/Cons
      - Exercise 1: Represent the list (a b c) graphically




      - Exercise 2: Graph the list (a (b c) d)

# ABSTRACT TYPE LIST

- Builder
  - cons e.g.: (cons 'a '(b)) => (a b)
- Selectors:
  - car e.g.: (car '(a b)) => a
  - cdr e.g.: (cdr '(a b)) => (b)
- Exercise: representing the data structures of parameters and results
- There are complex selectors that apply to lists. Example:
  - (cadr '(1 b c d)) b
  - (cddr '(1 b c d)) (c d)
  - (caddr '(1 b c d)) c
  - (cddddr '(1 b c d)) (d)

# MACROS

- Special forms of LISP that define extensions of the syntactic structure of the language.
- Unlike functions, they do not involve invoking and using the *stack*. They are mere lexical replacements.
- Example:
  - (first '(a b c d)) (car '(a b c d)) a
  - (rest '(a b c d)) (cdr '(a b c d)) (b c d)
  - (second '(a b c d)) (cadr '(a b c d)) b
  - (third '(a b c d)) (caddr '(a b c d)) c
  - ...

# BNF NOTATION

**BNF = Backus-Naur Form**

- Terminal Symbols
  - They belong to the language being described and allow you to construct expressions.
- Non-terminal symbols
  - They belong to the meta-language and are instantiated when constructing an expression.
- Production Rules
  - They define the structure of non-terminal symbols in terms of other non-terminal symbols and terminal symbols.
- BNF notation symbols

  | "alternative" a | A

  <> "non-terminal symbol delimiters"

  ::= "is described by:" <Var>::= a | A

  * "zero or more occurrences" {prmt}*

  + "one or more occurrences" {prmt}+

# EXAMPLE: SYNTACTIC DEFINITION OF TYPE **LIST** USING BNF

<List> ::= ( {<symbolic expression>}* ) | nil

<symbolic expression> ::= <List> | <Atom>

<Atom> ::= <number> |
< symbol> |
<boolean> |
<carater> |
< string> |
<other>

# ABSTRACT DATA TYPE

- Mathematical model
- Data Type ≠ of Data Structure

- Specifying a Data Type by
  - Identification of its possible values and the
  - Identify and define the **operations** that can be performed on this data.

- An abstract data type is implementation independent.

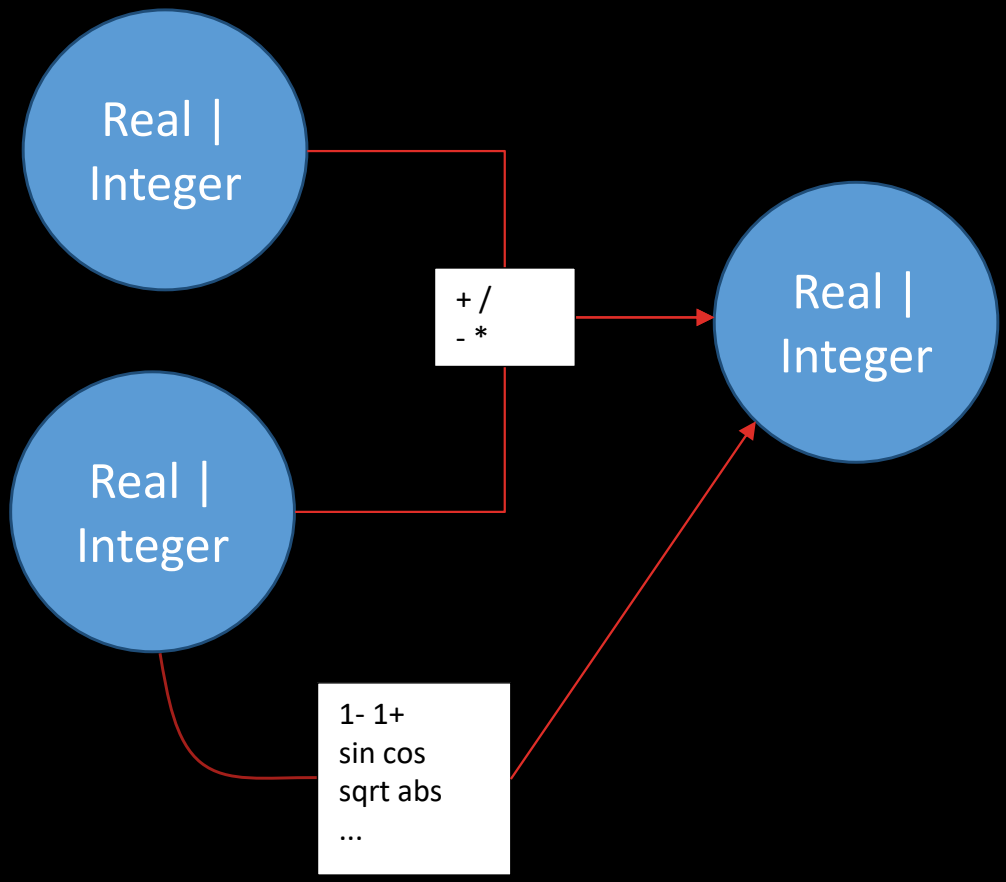
# ABSTRACT TYPE LIST: SOME MANIPULATION FUNCTIONS

list eg: (list 'a 'b) => (a b)
append ex.: (append '(a b) '(c d)) => (a b c d)
length ex.: (length '(a b)) => 2
nth e.g.: (nth 1 '(a b c d)) => b
reverse e.g. (reverse '(a b c d)) => (d c b a)
concatenate e.g.: (concatenate 'list '(a) '(b c))
=> (a b c)
Etc...

- Exercise:
  - Write the syntax of the **append** function invocation in BNF, knowing that it can have any number of args of type list.
  - identify ten existing list manipulation functions in LISP and write their syntax in BNF

# NUMERIC TYPES ARITHMETIC FUNCTIONS

(+
(-
(*
(/
(mod
(sqrt
(abs
(1+
(1-
(sin
(cos
(exp
(log
(random



# NUMBER OPERATIONS

- (/ 3 5) 3/5
- (1+ (/ 3 5)) 8/5

- (/ 3.0 5) 0.6
- (div 3 5) Error: Undefined operator DIV

- (mod 5 4) 1
- (mod 5.1 4) 1.099999
- (% 4 5) Error: Undefined operator % (% 4 5)

- (sqrt 9) 3.0

- (sqr 3) Error: Undefined operator SQR

# SOME INTERESTING FUNCTIONS: LOG, EXP AND RANDOM

(exp <n>)

power <n> of the Euler number (e)

Example: (exp 1) = 2.71828183

(log <n>)

Neperian (or natural) logarithm: base 2.71828183

and $^{x}$ = n (log 2.71828183) = ? (log 1) = ?

(log <n> <base> ) *; the base is an optional argument.*

Logarithm of base different from Euler's

Example: (log 8 2) = ?

(random < *number*> )

**random** accepts a number n and returns a random number of the same type (integer or real) between 0 (inclusive) and n (exclusive).

# VERY LARGE NUMBERS

- Integer type (numeric / list): BIGNUM

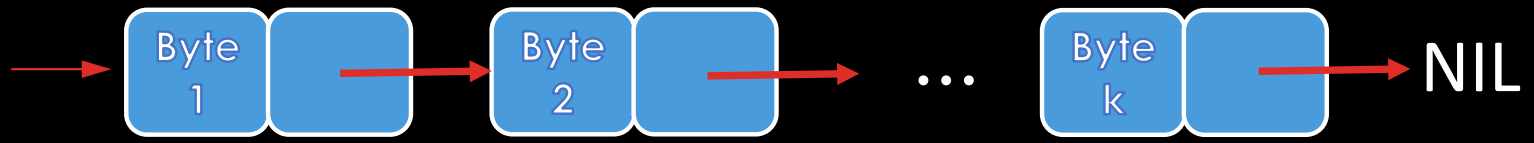
  (factorial 3) = 6

  (factorial 6) = 720

  (factorial 1000) = ??

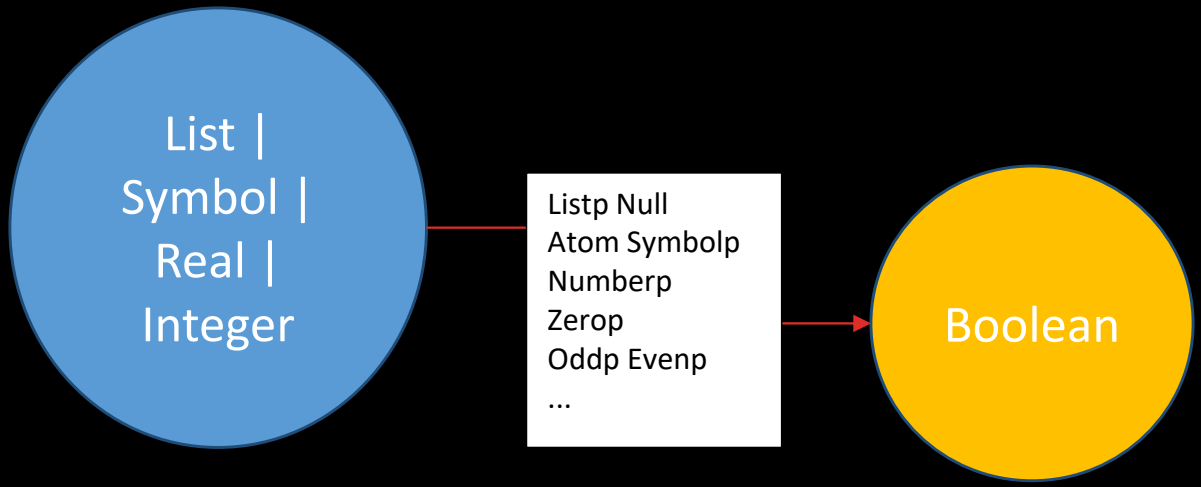
LISP gives all digits!

How? ... use lists:



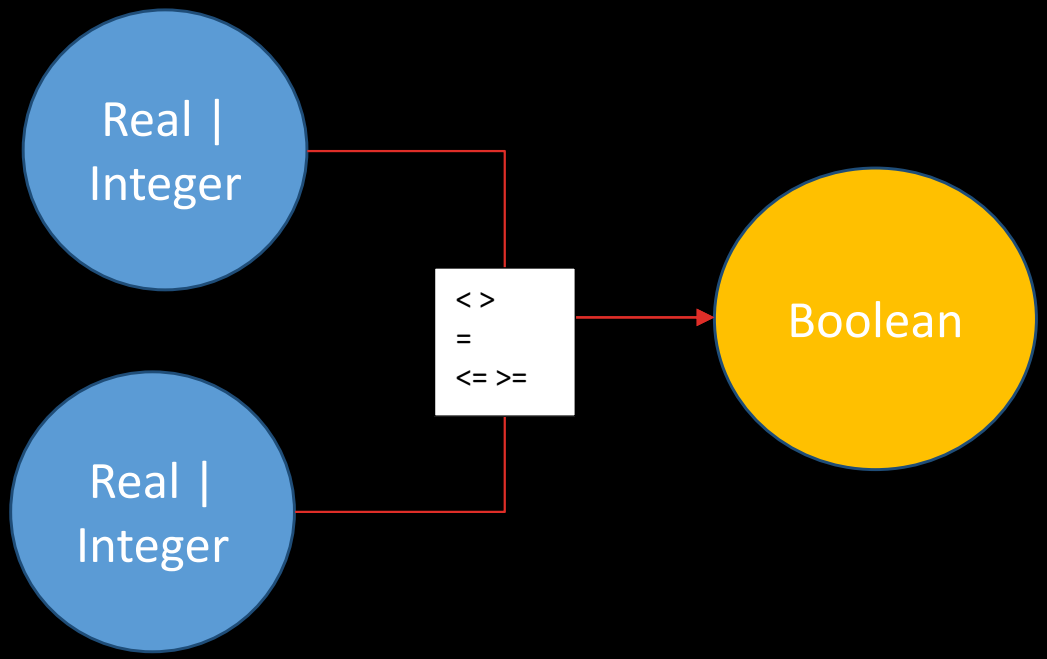
# TYPE BOOLEAN / PREDICATE

(null
(atom
(listp
(symbolp
(numberp
(zerop
(oddp
(evenp
(floatp
(integerp
(bignump



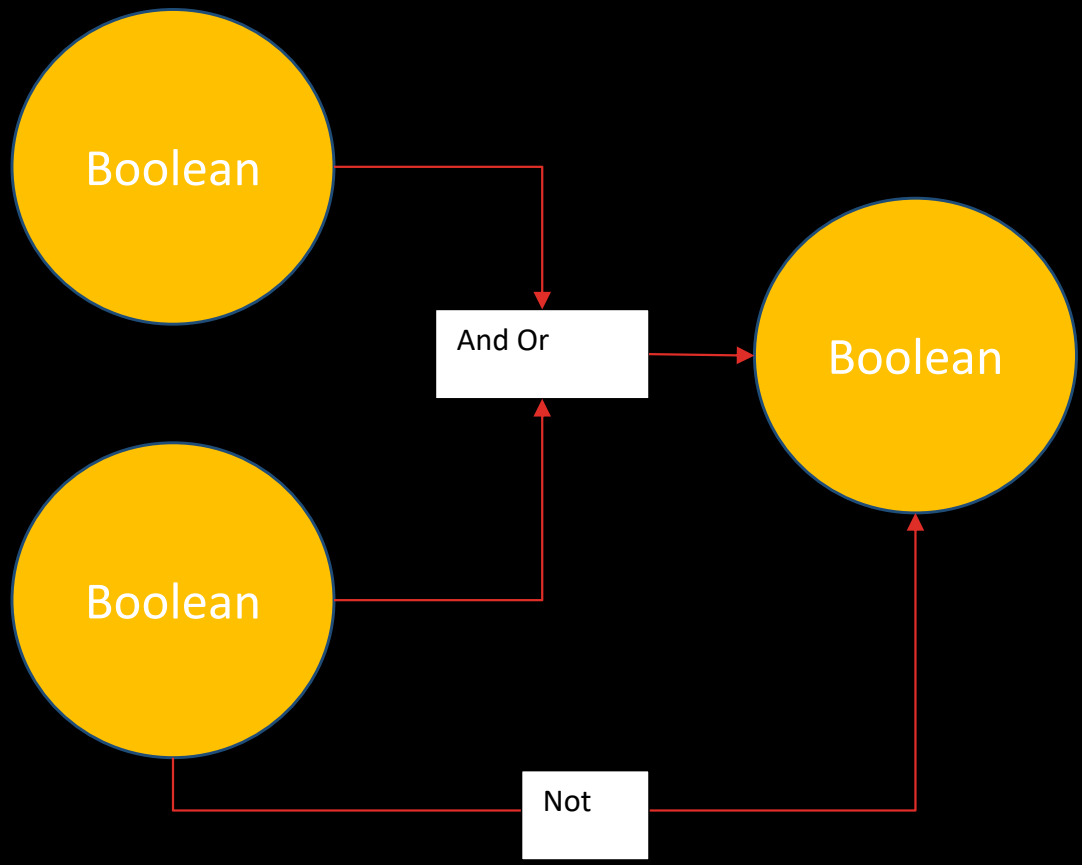
# RELATIONAL OPERATORS

(>
(<
(=
(>=
(<=



# TYPE BOOLEAN LOGIC OPERATORS

- and
- or
- not



# DETAILS FOR BOOLEAN EQUALITY OPERATORS

The difference between these functions is in the form of representation/data type; ref vs. value.

(equal
(eq
(eql
(=

```
(equal '(a b) '(a b)) T (eq '(a b) '(a b)) NIL
                                    (eql '(a b) '(a b)) NIL

(= 987654321 987654321) T (eq 987654321 987654321) NIL
(eql 987654321 987654321) T
(equal 987654321 987654321) T

(eq #\a) T
(eql #\a) T
(equal #\a #\a) T
```

```
> (bignump 987654321)
T
```

# EXERCISES

- From the list (3 4 "3" T 5) tell how many of its elements are and what kind they are (atom/list).
  - Make a diagram of the list.
- From the list (3 (4 6) nil () ((5)) ) tell what its elements are and what kind they are (atom/list). What is the fifth element?
  - Make a diagram of the list.
- What is the result of evaluating the expression (+ 3 (* 7 6) 5 (/ 4 2))?
- Which lists correspond to arithmetic expressions
  - (3+5+6+2)-1, (3+5+6)+2-1, 3+5+6+(2-1), 3+2*5, (5-(2-4))/(3*7) ?
- What is the tail of the list (3 4 5)?
  - And from the list ((3) (4 5))? What about the list ((3) (4 5))? What about the list containing nil? And from the list containing nil?

# EXERCISES

- What is the head of the tail of the following lists:
  - (), (3), (3 4), ((3) 4), ((3) (4 5)), ((3) (4 5) (7))
- Name the result of the following expressions:
  - (cons 1 nil), (cons 3 '(5)), (cons '(3) '(2)), (cons '(3) ()), (cons '(3 (5 6)) '((2 4) 6) )
- Name the symbolic expression containing the maximum number of **cons** that allows you to construct the list (3 (4) 5). What about the list (3 ((4) 6) 5) ?
- Name the expression containing **car**'s and **cdr**'s that allows you to get 5 from the list (3 4 5 6). And from the list (3 4 (1 3 (6 5)) (1 2))?
- Tell us the difference between the evaluation of
  - (car '((1) 2)) and from (car '(car ((1) 2)))
  - (cons (+ 1 2) '(3)) and from (cons '(+ 1 2) '(3))

# DEFINITION OF FUNCTIONS AND CONTROL STRUCTURES

# ROLE DEFINITION

```
(defun <function name> (< args>)
   "<description>"
   <function body>)
```

```
;;;; Example:
(defun average (note1 note2 note3)
  "takes the arithmetic mean of three notes"
   (/ (+ note1 note2 note3) 3)) returns a number
```

# EXAMPLES

- Function to calculate the square of a number

```
(defun square (number)
  "returns the square of a number"
  (* number number number))
```

- Function for calculating the area of a circle

```
(defun areaCircle (radius)
  "returns the area of a circle, given the radius"
  (* 3.14159265 (radius square)) ;pi is not preset
```

# FUNCTIONS AND FUNCTION SYMBOLS

- Functions with a name.
  - E.g.: (defun sum3 (x) (+ x 3))
    - Symbol "sum3" has in the slot ***function*** this definition.
- Functions without a name.
  - E.g.: (lambda (x) (+ 3 x))
    - There is no symbol to memorize this setting.
- What is executable is the definition (lambda)
- Example of using lambda functions in LISP:
  - ( **(lambda(x) (+ 3 x))** 7) => 10
  - (* 2 (**(lambda(x) (+ 3 x)) 7**) => 20

| **sum3** |
| --- |
| Name: "sum3 |
| Function: (lambda(...) |
| Value: nil |
| PList: nil |
| Package: default |

# DEBUGGING

(trace {<function>}*) - indicates the values of the arguments and the result each time a function is invoked.

(dribble <file>) - sends the output to the screen and to a file, simultaneously

(describe <function>) - gives the parameters and documentation if any

# CONTROL STRUCTURES OF PURE LISP

- Sequencing:
  - it doesn't.
- Selection:
  - Cond
- Repetition:
  - uses recursion

# SELECTION

- Syntax:

(cond <clause 1>
<clause 2>
...
<clause n> )

<clause> ::= (<condition> <result>)

# COND: EXAMPLES

```
(cond ((oddp x) 'odd)
      ((evenp x) 'even))

(cond ((oddp x) 'odd)
      (t 'par))

(cond ((= x 0) 1)
      (t (* x (fact (1- x)))))
```

# MACROS

- if

  (if <condition> <if-true> <if-false>)

  example: (if (zerop 3) "null" "ok") "ok

- ecase

  (ecase <key> (<item> <val>) .... (<item> <val>))

  example: (ecase b (a 1) (b 2) (c 3)) 2

# RECURSIVITY

- A recursive function always has a structure with 2 conditions or more:
  - Stop condition
  - Recursive condition (1 or more)

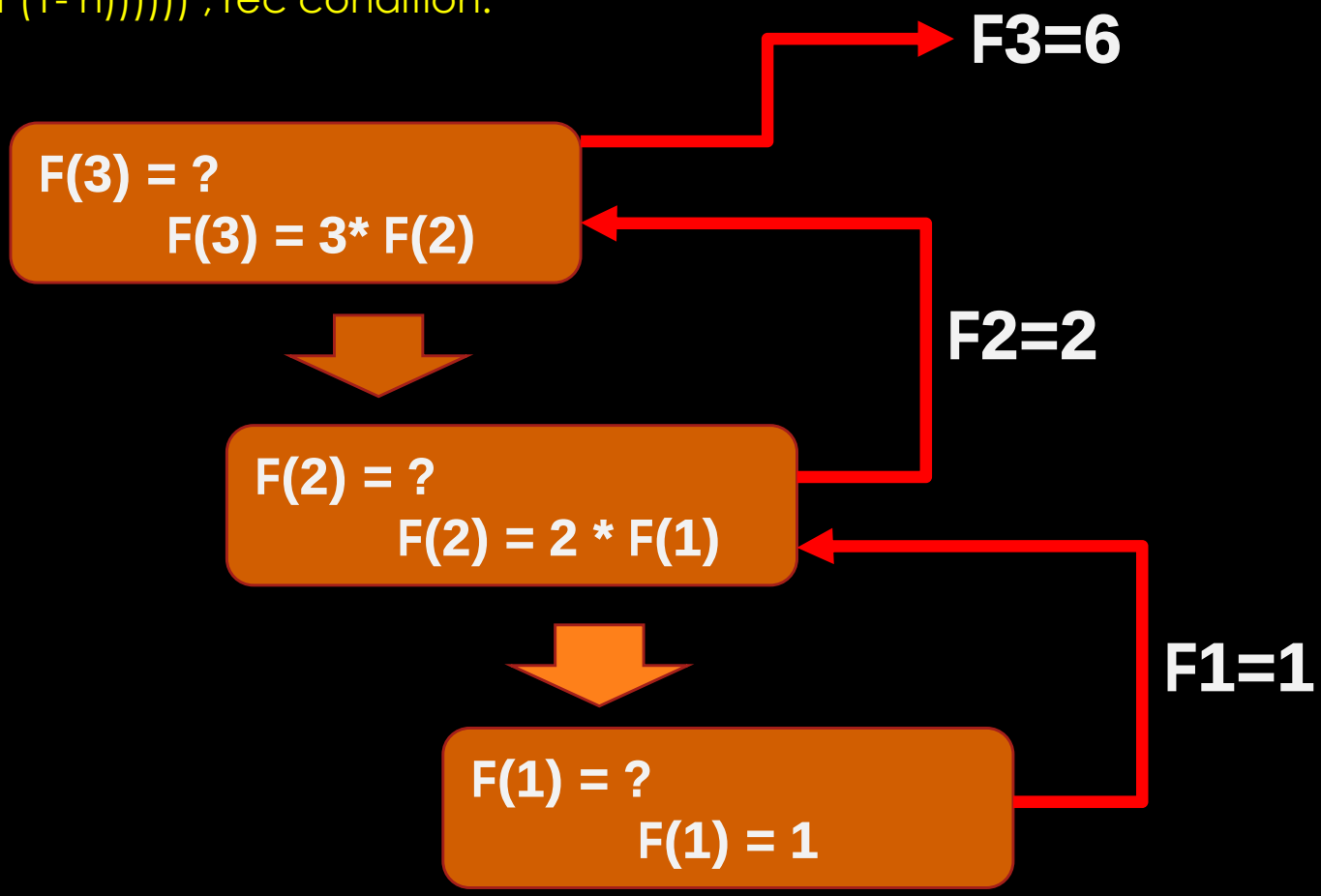
Example:

```
(defun f (n)
   (cond ((<= n 1) 1) ; stop condition
         (T (* n (f (1- n)))))) ; recursive condition
```

What does this function do?

# USING THE STACK

```
(defun f(n)
    (cond ((<= n 1) 1) ; stop condition
          (t (* n (f (1- n)))))) ; rec condition.
```



# EXERCICIOS

- Define a **pot** function to calculate: pot(x,n)=$x^n$
- Define a **fib** function to calculate the Fibonacci sequence:

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

- Define the Ackermann function **A**:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

# USE WITH EXTREME CAUTION AND ONLY IN EXCEPTIONAL CASES

- Sequencing

  (progn <$expr_1$ > <$expr_2$ > ... < $expr_n$ >)

- Iteration

  **(**do *({<var> | (<var> [<init-form> [<step-form>]])}*)*

  *(< end-test-form> < result-form>* ***)***

  *< declaration>* ****** *{< tag> | < statement>}**

  *<body>)*

  dotimes

  dolist

# EXAMPLES

```
(do ((temp-one 1 (1+ temp-one))
     (temp-two 0 (1- temp-two)))
    ((> (- temp-one temp-two) 5) temp-one)) 4
```

Variant:

(dotimes (<counter> <limit> [<result>]) <body>)

E.g.: (dotimes (i 10) (progn (princ i) (terpri))

(dolist (< var> <list> [<result>]) <body>)

E.g.: (dolist (x '(a b c)) (print x))

# GLOBAL VARIABLES AND CONSTANTS

- To be used with extreme caution and only in very special cases.

  (defparameter <var> <value>)

- These variables are outside of any lexical environment so they are visible throughout the program, at the risk of causing unpredictable side effects.
- Generally the convention of naming them with asterisks on the left and right is used.
  - Example: (defparameter *pi* 3.14159265)

# LINKING VALUES TO VARIABLES IN LEXICAL ENVIRONMENTS

- LET - Evaluation of expressions is done in parallel.
  (let ({(<var> < expr>)}*)
    <let body>)

- LET* - Sequential evaluation of exprs.
  (let* ({(<var> < expr>)}*)
    <let body>)

# LET EXAMPLE

- Define a function to calculate the perimeter and area of a circle of radius given by the user. Returns a list with these two values.

```
(defun p-a-circle ()
    (let ((radius (read)))
      (list (* 2 3.14 ray) (* 3.14 ray)))) ray
```

# EXAMPLE LET*

- Define a RA10 function that returns the square root of the amplitude of a list of numbers if the amplitude is greater than 10, or zero otherwise.
- The value range of a list of numbers is the difference between maximum and minimum: (max - min)

```
(defun RA10 (list)
    (let* ((largest (max list)))
           (smaller (min list))
           (amplitude (- major minor))
      (cond ((< amplitude 10) 0)
            (t (sqrt amplitude)))))
```

# LET AND LAMBDA

(let ((a$_1$ b$_1$ ) (a$_2$ b$_2$ ) ... (a$_n$ b$_n$ ))
(body to$_1$ to$_2$ ... to$_n$ ))

*It is similar to:*

((lambda (a$_1$ a$_2$ ... a$_n$ )
(body to$_1$ to$_2$ ... to$_n$ ))
b$_1$ b$_2$ ... b$_n$ )

# LET* AND LAMBDA

(let* ((a$_1$ b$_1$ ) (a$_2$ b$_2$ ) ... (a$_n$ b$_n$ ))
  (body to$_1$ to$_2$ ... to$_n$ ))

It is similar to:

((lambda (a$_1$ )
  ((lambda (a$_2$ )
    ...
    ((lambda (a$_n$ )
      (body to$_1$ to$_2$ ... to$_n$ ))
     b$_n$ ))
   b$_2$ ))
 b$_1$ )

# LINKING FUNCTIONS TO VARIABLES IN LEXICAL ENVIRONMENTS

- While functions are defined globally with defun, you can also create local functions with flet and labels.

```
(flet (function-definition*)
  body-form*)

(labels (function-definition*) ;;;; allows recursive definitions
  body-form*)
```

# EXAMPLES OF FLET AND LABELS

```
(defun test (x)
   (flet ( (f1 (n) (+ n n)) )
      (flet ( (f2 (n) (+ 2 (f1 n)) ) ;;;; there is no flet*
         (f2 x)))
=> TEST
(test 5) 12

(defun recursive-times (k n)
   (labels ( (temp (n)
                  (if (zerop n) 0 (+ k (temp (1- n))))) )
      (temp n)))
=> RECURSIVE-TIMES
(recursive-times 2 3) 6
```

# CLOSURES

- A lexical *closure* is a function that, when invoked with arguments, executes the body of a lambda expression in the lexical environment that was captured when the *closure was* created augmented with the function's parameter bindings to its arguments.

```
(let ((e 1))
  (defun clos-1 (x) (+ x e))

> (clos-1 3) 4
> and unbound variable
```

What is *scope*?
- lexicon
- dynamic

and life span?

# SOME DESTRUCTIVE FUNCTIONS USE ONLY IN LEXICAL ENVIRONMENTS

setf : assignment. Example: (setf x 3)

incf : destructive increment: (incf x)

```
> (let ((counter 0))
    (defun counter-next ()
      (incf counter))
    (defun counter-reset ()
      (setf counter 0))

> (counter-next) 1
> (counter-next) 2
> (counter-next) 3
> (counter-reset) 0
> (counter-next) 1
```

# ANOTHER EXAMPLE

```
> (let ((password nil))
        (secret nil))
   (defun set-password (new-passwd)
    (if password
     "the password is already set"
     (setf password new-passwd)))
   (defun change-password (old-passwd new-passwd)
    (if (eq old-passwd password)
     (setf password new-passwd)
     "Password not changed"))
   (defun set-secret (passwd new-secret)
    (if (eq passwd password)
     (setf secret new-secret)
     "Wrong password"))
   (defun get-secret (passwd)
    (if (eq passwd password)
     secret
     "Did you want..."))))
```

```
> (get-secret 'sesame) "Did you want..."
> (set-password 'valentine)
> (set-secret 'sesame 'my-secret) "Wrong password"
> (set-secret 'valentine 'my-secret) MY-SECRET
> (get-secret 'fubar) "Did you want..."
> (get-secret 'valentine) MY-SECRET
> (change-password 'fubar 'new-password)
                "Password not changed"
> (change-password 'valentine 'new-password)
                NEW-PASSWORD
> (get-secret 'valentine) "Did you want..."
> Password Error: unbound variable
> Secret Error: unbound variable
> (setf password 'cheat)
> (get-secret 'cheat) "Did you want..."
```

# EXAMPLE 2

The previous example is only for keeping 1 secret because each time you evaluate the LET you redefine all the functions (*closures*).

Possible Solution:

```
> (defun make-secret-keeper ()
    (let ((password nil)
          (secret nil))
      #'(lambda (operation & rest arguments)
          (ecase operation
                  (set-password (let ((new-passwd (first arguments)))
                                  (if password "the password is already set"
                                    (setf password new-passwd))))
                  (change-password (let ((old-password (first arguments))
                                         (new-passwd (second arguments)))
                                     (if (eq old-passwd password) (setf password new-passwd)
                                       "Password not changed")))
                  (set-secret (let ((passwd (first arguments))
                                    (new-secret (second arguments)))
                                (if (eq passwd password) (setf secret new-secret)
                                  "Wrong password"))))
                  (get-secret (let ((passwd (first arguments)))
                                (if (eq passwd password) secret
                                  "Did you want...")))))))
```

# EXAMPLE 2 (CONT.)

```
> (defparameter secret-1 (make-secret-keeper))
> secret-1 #<LEXICAL-CLOSURE #x36AE056>
> (funcall secret-1 'set-password 'valentine) VALENTINE
> (funcall secret-1 'set-secret 'valentine 'deep-dark) DEEP-DARK

> (defparameter secret-2 (make-secret-keeper))
> (funcall secret-2 'set-password 'bloody) BLOODY
> (funcall secret-2 'set-secret 'bloody 'mysterious) MYSTERIOUS
> (funcall secret-2 'get-secret 'valentine) "Wrong password"

> (funcall secret-1 'get-secret 'valentine) DEEP-DARK
```

# EXERCISES

- Define roles for
  - Rotate a list to the left.
    - (wheel-left '(1 2 3 4)) => (2 3 4 1)
  - Calculate the real roots of the 2nd degree equation.
    - (fresolvent 1 0 -1)=> (-1 1)
- Define the following recursive functions:
  - Maximum: max(number list)=>maximum value
  - Fibonacci: fib(n) => n-th in the Fib sequence.
  - Binary: bin(n)=(1 0 ... 0) which, given a positive integer, constructs a list of zeros and ones, forming the binary representation of the argument.
  - Hexadecimal: hex(n)=(0 A 3 ... F) which, given a positive integer, constructs a list of symbols between 0 and F, forming the hexadecimal representation of the argument.

# EXERCISES

- Write functions to solve the following problems:
- Mirror(L): Receives a list, which may have sublists, and inverts the order of all elements, including those within the sublists
- List-ate-n(n): Takes a number and returns a list of all integers up to n sorted in ascending order.
- Insert(e p L): inserts an element e at position p in a list.
- Insert-ordered(n L-ord): inserts a number into a list that is already sorted in ascending order, so that the result remains an ordered list, using as few operations as possible (binary search).
- Alisa(L): Takes a list with sublists and returns a list with the elements in the same order but without sublists.

# SET EXERCISES

A set is assumed to be represented by a list of elements without repetition.

- Set(List): Takes a list and returns a set (eliminates repeats)
- Intersection(C1 C2): Receives two sets and returns the intersection set.
- Reunion(C1 C2): Receives two sets and returns the meet set.
- Difference(C1 C2): Receives two sets and returns the set difference (C1 - C2).
- Add(e C1): Returns the set that results from adding e to set C1.
- Subtract(e C1): Returns the set that results from taking e from the set C1.

# LAMBDA FUNCTIONS, PARAMETERS, AND META-FUNCTIONS

# LAMBDA

- The lambda function is the basic programming element in LISP.
- It follows from the lambda calculation
- Example:
  - Definition of function: (lambda (x y) (+ x y 1))
  - Invocation: ((lambda (x y) (+ x y 1)) 2 5) 8

# FULL SYNTAX OF PARAMETERS

- (lambda ({var}*
  [&optional {var | (*var* [*initform [svar*])}*]
  [&rest *var]*
  [&key {var | ({var | (*keyword var*)} [*initform [svar*]])}*
  [&allow-other-keys]]
  [&aux {var | (*var* [*initform*])}*)]
  **[[**{declaration}* | *documentation-string***]]**
  {form}*)

  The var and svar must be symbols

  When the function is invoked, the keywords must start with the character :

  An initform can be any LISP expression.

# EXAMPLES

- &rest
  - (defun g (x &rest r) ...)
  - (g 1 2 3 4 5) ... x=1 r=(2 3 4 5)

- &optional
  - (defun f (a &optional (b 5))
    (+ a b))
  - (defun f (a &optional (b 5) &rest z)
    (cons a (cons b z)))

# EXAMPLES (CONT.)

- &key
  - (defun f (x &key y z w) ...)
  - (f 1 :w 3 :y 1) ... x=1 y=1 z=<unbound> w=3

  - (defun f (x &key y (z 0) w) ...)
  - (f 1 :w 3 :y 1) ... x=1 y=1 z=0 w=3

# FUNCTIONAL (META-FUNCTIONS)

- Direct access to the Lisp evaluator (same as used in the REPL)

(eval < form>)

*example*: (defun f(n)
(eval (cons (nth n '(+ - / *)) '(2 3))))

(apply <function> <list prmts>)

*example*: (defun g(n)
(apply (nth n '(+ - / *)) '(2 3))

(funcall <function> < prmts>)

*example*: (defun h(n)
(funcall (nth n '(+ - / *)) 2 3))

# CORRESPONDENCE FUNCTIONS

**mapcar** *function list* & rest *more-lists*
**maplist** *function list* & rest *more-lists*

**mapc** *function list* & rest *more-lists* -- do not accumulate the result
**mapl** *function list* & rest *more-lists*

**mapcan** *function list* & rest *more-lists* -- use destructive functions
**mapcon** *function list* & rest *more-lists*

- Examples:

```
(mapcar '+ '(1 2 3) '(6 7 8)) => (7 9 11) ;; 2-argument function
(mapcar #'abs '(3 -4 2 -5 -6)) => (3 4 2 5 6) ;; 1-argument function

(maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
                                => ((foo a b c d) (foo b c d) (foo c d) (foo d))
```

# EXAMPLE

- Given a list of triplets with the following structure:

  <person> ::= (<name> <age> <country>)

  <persona> ::= (<persona>*)

Define a function to list the countries of people over 50.

```
(defun larger50 (people)
  (apply #'append ;; removes nils from a list of lists
    (mapcar (lambda (person)
              (cond ((> (second person) 50) (cddr person)))
                    (t nil) ))
            people))))
```

;;;; (larger50 '((antonio 51 EN)(brad 34 UK) (charles 77 FR))) >> (EN FR)

# DETAILS

- Apply #'append ...
  - Useful for removing nils from a list of lists.

```
(apply #'append '((a) (b) nil (c) nil nil (d)))
>> (a b c d)
```

Detail:

```
( (lambda (person) (cond ((> (second list) 50) (third p)) (t nil)) '(c 77 EN) )
>> EN

( (lambda (person) (cond ((> (second list) 50) (cddr p)) (t nil))) '(c 77 EN) )
>> (PT)
```

# EXERCISES

- What is the result of each of the following function invocations?
  - (mapcar #'1+ '(1 2 3 4 5))
  - (mapcar #'list '(1 2 3 4 5))
  - (mapcar #'list '(1 2 3 4 5) '(a b c))
- Write a function to add 2 units to all elements of a given list of numbers.
- Write a function to sum the square roots of a list of numbers.
  - Recursive version
  - Version using higher-order functions
- Write a function to count the number of odd numbers in a list.

# EXERCISES

- Define roles for
  - 1) multiply two two-dimensional matrices NxM and MxK
    (defun multimap (m1 m2)
  - 2) transpose a square matrix
    (transposed defun (m)

The number of rows and columns in each matrix is arbitrary.

Use a data structure in the form of a list of lists as a representation of the matrix data type.

```
( ( ... )
  ( ... )
    ... )
```

# I/O FUNCTIONS

# I/O FUNCTIONS

- Basic
  - Read
  - Read-line
  - Write-Line
  - Terpri
  - Format
  - ...

- These functions have an optional argument which is the <stream> or <port>.
  - T = (*standard-io*) = screen
  - NIL = string

# READING

- READ

  Arguments:

  - (&OPTIONAL
    (STREAM *STANDARD-INPUT*)
    (EOF-ERROR-P T)
    EOF-VALUE
    RECURSIVE-P)
  - Reads the next value in the stream (defaulted to *standard-input*)

    - EOF-ERROR-P and EOF-VALUE specify what happens if the program tries to read from an empty stream:
      - If EOF-ERROR-P has the value true then Lisp generates an exception. Otherwise it returns EOF-VALUE (default nil).
    - RECURSIVE-P is reserved for functions invoked by the Lisp reader.

# READING (CONT.)

- READ-LINE

  Arguments:

  - (&OPTIONAL
    (STREAM *STANDARD-INPUT*)
    (EOF-ERROR-P T)
    EOF-VALUE
    RECURSIVE-P)
  - Returns the line of text read from the stream in the form of a string, discarding the newline.

- Other
  - READ-BYTE *stream &optional eof-error-p eof-value*
  - READ-CHAR *&optional stream eof-error-p eof-value recursive-p*

# WRITING

- WRITE-LINE

  Arguments: (STRING &OPTIONAL
  (STREAM *STANDARD-OUTPUT*)
  &KEY (START 0) END)

  Writes a String to the given Stream, followed by a newline.

- TERPRI

  Arguments: (&OPTIONAL
  (STREAM *STANDARD-OUTPUT*))

  Write a newline in the indicated stream.

# WRITING

- PRINC

  Argument0s:

  (OBJECT &OPTIONAL (OUT-STREAM *STANDARD-OUTPUT*))

  Write an aesthetically pleasing representation of the object in the given stream, but not necessarily READable.

  Example: (princ "abc") >> abc

- PRIN1

  Arguments:

  (OBJECT &OPTIONAL (OUT-STREAM *STANDARD-OUTPUT*))

  Write a representation of the object to the given stream, which is generally READable.

  Example: (princ "abc") >> "abc"

# WRITING

- FORMAT

  Arguments:

  (STREAM CONTROL-STRING &REST FORMAT-ARGUMENTS)

  - If STREAM= T, the result is written to the *standard-output*,
  - If it is NIL, the result is returned as a string.
  - Otherwise, STREAM usually corresponds to a file.

  - The CONTROL-STRING contains the string to write, usually with built-in directives, starting with the character "~".

# FORMAT

GUIDELINES:

~A or ~nA Write an argument as PRINC
~S or ~nS Write an argument as PRIN1
~D or ~nD Write an argument as an integer
~% TERPRI
~& FRESH-LINE

n is the width of the field in which the value is written

Examples:

- Real numbers:
  - (format t "~$" pi)
    - 3.14
  - (format t "~5$" pi)
    - 3.14159

Whole numbers:
(format t "~d" 1000000)
1000000
(format t "~:d" 1000000)
1,000,000
(format t "~@d" 1000000)
+1000000

# FORMAT (CONT.)

More examples:

- (format nil "The value is: ~a" 10)
  - "The value is: 10"
- (format nil "The value is: ~a" "foo")
  - "The value is: foo"
- (format nil "The value is: ~a" (list 1 2 3))
  - "The value is: (1 2 3)"

Format Conditional:

- (format nil "~[zero~;one~; two~]" 0) ==> "zero"
- (format nil "~[zero~;one~; two~]" 1) ==> "one"
- (format nil "~[zero~;one~; two~]" 2) ==> "two"

# MORE DIRECTIVES

- ~% new line
- ~& fresh line
- ~| page break
- ~T tab stop
- ~< justification
- ~> terminate ~<
- ~C character
- ~( case conversion
- ~) terminate ~(
- ~D decimal integer
- ~B binary integer

- ~O octal integer
- ~X hexadecimal integer
- ~bR base-b integer
- ~R spell an integer
- ~P plural
- ~F floating point
- ~E scientific notation
- ~G ~F or ~E, depending upon magnitude
- ~$ monetary
- ~A legibly, without escapes
- ~S Readably, with escapes
- ~~ ~

# EXERCISES

- Write a function called **write_number** that takes a number and writes to the screen "The number is ...". Use the format function.
- Write a function called **sequencia_numeros** that reads a sequence of positive numbers via the keyboard, in any order, ending with zero, and writes to the screen the sequence of numbers read in ascending order, one number on each line.
- Write a function called **write_list** that writes the contents of a list to the screen.
  - (write_list '(1 2 3))

    (1 2 3)
- Redefine the previous question in **write_list1** so that you can write one **list** element per line. The use of the mapc function is recommended.
  - (write_list1 '(1 2 3))

    1
    2
    3

# FILES

- open *filename*
  & key
  :direction
  :element-type
  :if-exists
  :if-does-not-exist
  :external-format

- (with-open-file (<port> <path> {keys}*) <body>)

# KEYWORDS FROM OPEN

- Keyword Value Stream Direction ---------- :DIRECTION
  - :INPUT input (default)
  - :OUTPUT output
  - :IO input & output
  - :PROBE none
- Keyword Value Action if File Exists ---------- :IF-EXISTS
  - :ERROR signal an error
  - :NEW-VERSION next version (or error)
  - :RENAME rename existing, create new
  - :SUPERSEDE replace file upon CLOSE
  - :RENAME-AND-DELETE rename and delete existing, create new
  - :OVERWRITE reuse existing file (position at start)
  - :APPEND reuse existing file (position at end)
- Keyword Value Action if File Does Not Exist ---------- :IF-DOES-NOT-EXIST
  - :ERROR signal an error
  - :CREATE create the file

# OPEN KEYWORDS (CONT.)

- Keyword Value Element Type ---------- :ELEMENT-TYPE
  - :DEFAULT character (default)
  - 'CHARACTER character
  - SIGNED-BYTE signed byte
  - UNSIGNED-BYTE unsigned byte
  - *other* implementation-dependent
- Keyword Value File Format ---------- :EXTERNAL-FORMAT
  - :DEFAULT default
  - *other* implementation-dependent

# PATHNAME

In different operating systems the functions for accessing the file system can differ, and an intermediate level of abstraction is convenient by means of the "pathname" data type:

Builder:

make-pathname &key :host :device :directory :name :type :version :defaults :case

Selectors:

pathname-host pathname
pathname-device pathname
pathname-directory pathname
pathname-name pathname
pathname-type pathname
pathname-version pathname

Special access:

user-homedir-pathname & optional *host*

Test

probe-file namestring

Conversion to string:

namestring *pathname*

# EXAMPLE

```
(make-pathname
        :host "technodrome"
        :directory '(:absolute "usr" "krang")
        :name "shredder")
  => #P "technodrome:/usr/krang/shredder"
```

*Note: Under Windows, access to "c:\..." is restricted. Preferable to put the working files in a directory below the root or on another disk.*

# EXAMPLE

Write a function that copies the odd lines of a ".txt" file to a new file with the same name and ".odd" extension.

```
(defun copy-impares (file1)
       (let ((path1 (make-pathname :host "e" :directory '(:absolute "") :name file1 :type "txt")))
            (path2 (make-pathname :host "e" :directory '(:absolute "") :name file1 :type "odd")))
          (with-open-file (f1 path1 :direction :input)
            (with-open-file (f2 path2 :direction :output :if-exists :supersede)
              (copy-line f1 f2)))))


(defun copy-line (f1 f2)
     (let ((line (read-line f1 nil :end)))
       (cond ((not (eq line :end))(write-line line f2) (skip-line f1 f2))
             (t (close f2)))))


(defun line jumper (f1 f2)
     (let ((line (read-line f1 nil :end)))
       (cond ((not (eq line :end))(copy-line f1 f2))
             (t (close f2)))))
```

# EXERCISES

- Using the with-open-file function define a **write_list_file** function that gets a list and a full name (path + name) of a file and writes the contents of that list - with one element per line - to that file.
- Using the with-open-file function, and the read function define a **read_elements_file** function that takes a complete name from a file and reads all the elements in that file and writes them to the screen element by element, one on each line.
- Write a function called count_if that counts all characters in a file whose ASCII code is between a given lower and upper bound and returns a list of pairs where each pair consists of the ASCII code of the relevant character and its number of occurrences in the file.
    - You can use the functions: (code-char <num>) and (char-code < char>)
- Write a function called **transform_elements** to read the numeric elements from a given input file, and apply a **transform** function to each element, which converts the element to hexadecimal, and writes the result of applying that function to a given output file.

# PACKAGES

# THE PROBLEM

- Purpose: To avoid confusing symbols that have the same name but exist in different software modules.
  - Namespace problem
- The problem is solved at the Reader level in the REPL (read-eval-print loop).
- Two main possibilities:
  - Symbol occurrences with the same name in different modules must be the same.
  - Occurrences of symbols with the same name in different modules should be different.

# POSSIBLE SOLUTIONS

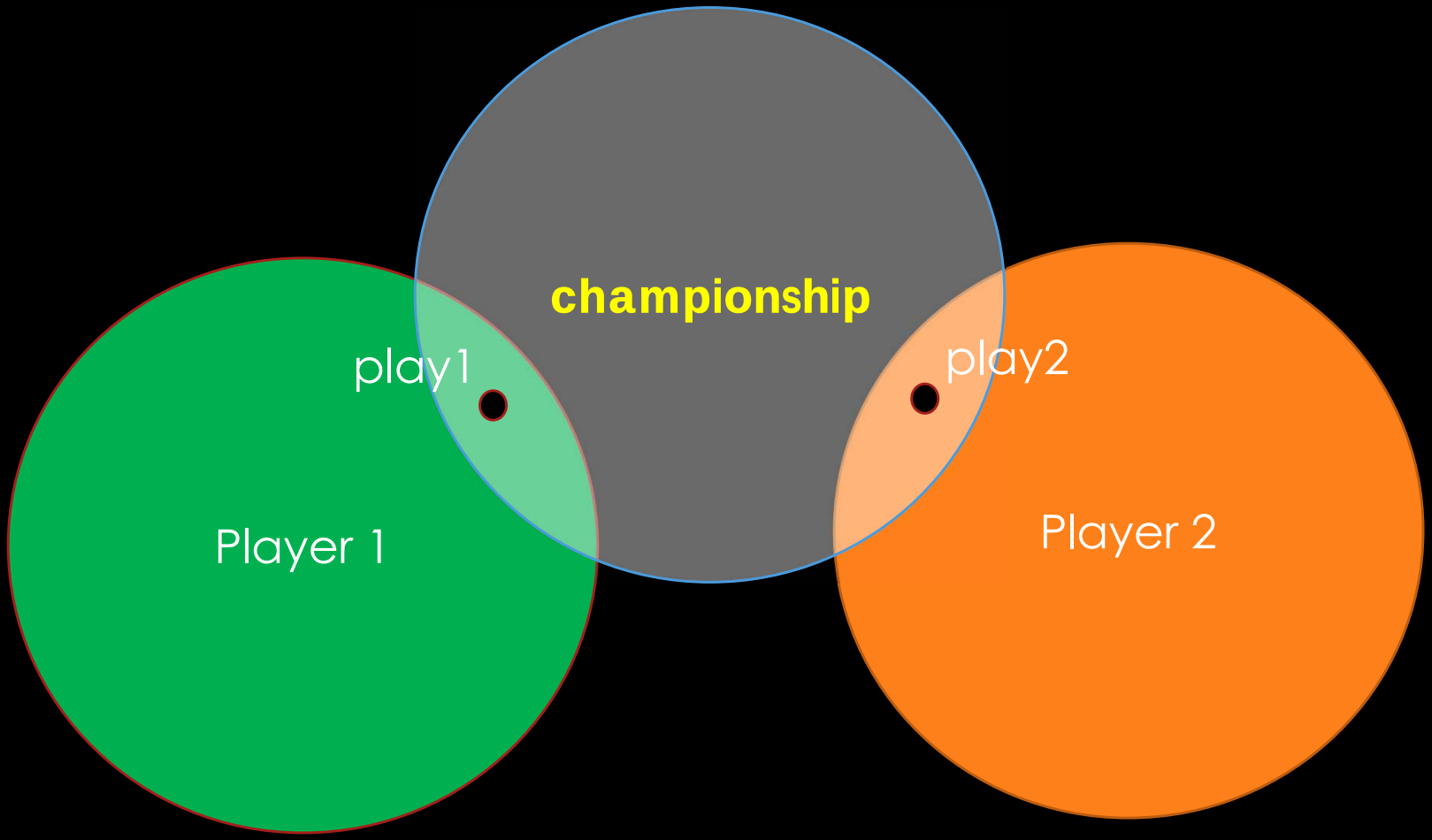
- In different modules use different prefixes for the symbols.
- Different programmers can build different modules without interfering with each other.

- Problems:
  - Name management depends on the programmer's self-discipline.
  - Difficulty matching that a name is the same symbol (with the same value and/or same function definition) in multiple modules.

# CONCRETE EXAMPLE

- Championship of a 2-player game
- Required:
  - Program to manage the game, invoking the main function of each player (receives the state and returns the move).
  - Program that determines player 1's move.
  - Program that determines player 2's move.
- The game management program has to use symbols that belong to each of the other programs (the play function)
- Each player's program must encapsulate all other symbols to avoid conflicts.

# EXAMPLE (SCHEMATIC)



# THE BEST SOLUTION: PACKAGES

- LISP has namespaces implemented in the form of packages (symbol packages)
- There are some pre-defined packages. Example:
  - Common-LISP (nickname: CL)
  - Common-LISP-User (nickname: CL-User)
    - The CL-User uses the CL
- There is a constant *package* whose value is the default package (CL-User)
- The definition of packages is done with defpackage

# SIMPLE USAGE EXAMPLE

```
(defpackage :player1)
(defpackage :player2)

(load "w:\\j1.lisp")
(load "w:\\j2.lisp")
```

# PROGRAM J1.LISP

```
(in-package :player1)

(defun alphabet (...)
        ...)
(defun successors (e)
        ...)
(defun play (x)
        ...)
```

# J2.LISP PROGRAM

```
(in-package :player2)

(defun alphabet (...)
        ...)
(defun successors (e)
        ...)
(defun play (x)
        ...)
```

# RECOMMENDATIONS

- Each file must indicate the package in which the symbols are defined with in-package
- There must be only one in-package per file and it must be the first line of the file.
- In order to use in-package you have to define the package first (defpackage).

# DEFINE PACKAGES

## (defpackage name & rest options)

Name = keyword, corresponding to the package name

options::=

(:nicknames nickname*)* |
(:documentation string) |
(:use package-name*)* |
(:shadow {symbol-name}*)* |
(:shadowing-import-from package-name {symbol-name}*)* |
(:import-from package-name {symbol-name}*)* |
(:export {symbol-name}*)* | |
(:intern {symbol-name}*)* | |
(:size integer)

# MAIN OPTIONS

- :use
  - set of packages that the package being defined inherits from. By default, if nothing is given, it takes on an implementation-dependent value - usually :Common-LISP.
- :import-from
  - Symbols defined as arguments are imported into the package being defined
- :export
  - Symbols defined as arguments are found or created in the package being defined and exported, to be shared with packages that use the package being defined

# EXPORT AND USE-PACKAGE

- Mechanism to import all relevant symbols from a given package:
  - All packages maintain a list of symbols that are supposed to be used by other packages, called the exported symbol list.
  - To add a symbol to this list you use the export function.
  - To remove a symbol from this list you use the unexport function.
  - To import all the symbols exported by a package we use the use-package function.
  - To undo the previous operation you use the unuse-package function.

# SHADOWING

- Shadowing symbols list
  - List associated with a package that contains the symbols exempt from "symbol conflict errors" detected when other packages are ":used
- :shadow
  - Set of symbols created in the package being defined that belong to the shadowing symbols list
- :shadowing-import-from
  - Set of symbols that are imported from the indicated package into the package being defined and that become part of the shadowing symbols list.
    - If there is a symbol with the same name in the package being defined that symbol is removed from the package.

# INTERN / UNINTERN

- All symbols that belong to a package are "interned".
- The symbols are interned in the current package, which is CL-User by default.
- The keywords are interned in a special package with the name KEYWORD.
- A symbol interned in a package can be uninterned, so it is no longer accessible. Example, if the current package has the factorial symbol:

```
> (unintern 'factorial)
T
```

# LIST OF SYMBOLS OF A PACKAGE

To get all the symbols defined in a package:

**do-symbols** *(var [package [result-form]]) declaration* {tag | statement}**

*There are 3 macros:*

DO-SYMBOLS, DO-EXTERNAL-SYMBOLS, DO-ALL-SYMBOLS

Example:

```
(do-external-symbols (s (find-package "KEYWORD"))
    (print s))
```

# EXERCISES

- Consider a 2-player game in which each player tries to get a number (goal) with a sequence of N rolls of a 6-sided die.
- The player who gets closest to the number wins.
- Make a program for player 1 to play. Use the package player-1.
- Make a program for player 2 to play. Use the package player-2.
- Make a program play in a package called "championship" that alternately plays the play functions exported by each of the packages "player-1" and "player-2".
  - The play functions take a state (in the form of a 3-element list: number to reach, player's own current number, and opponent's current number) and return a move (in the form of a boolean: t = play; nil=stop)
  - The league play function rolls the die (generates a random number) which adds to the next player, stopping the game when both players stop or when one of them exceeds the goal, in which case they lose.

# COMPLETE PROGRAMS

# ABSTRACT DATA TYPES

- LISP is an extensible language:
  - Allows you to define a more abstract language, better adapted to the application domain
    - Advantages:
      - Code comprehensibility
      - Reuse of complex code
      - Ease of changing the representation
  - => Bottom-Up Approach

# EXAMPLE 1

- Data type "Class
  Class is a group of students
  Need to set the type "Student

  Constructor of type Student (Student-new {:< prop> < val>}*)
  < prop> is the name of some property
  <val> the respective value
  at the very least you need the student's #.
  Selector of type Student (Student-< prop> <student>)

  Constructor of type Class (NewClass {<students>})
  Selector of type Class (Class-students {:< prop> < val>}*)

  It is usual for abstract types to have further operations:
  comparison, predicates, reading, writing, in addition to all the abstract type specific operations.

# EXERCISE 1

- Make a program to manage a staff of students in a class:
  - Add students to a class
    - By reading a file
    - Interactively
      - Each student can be represented by a list with: student number, name, project 1 grade, project 2 grade, exam grade and final average.
  - Modify a student's grade
  - Look for students with positive grades
  - Search for students whose name begins with a given letter
  - Calculating the mean and mode of the class grades
  - Calculate the histogram based on quartiles
  - Sort the staff by note or by name
  - Write the agenda
    - In a file
    - On the computer screen

# EXAMPLE 2

- Animal game: the computer asks the user questions to try to find out what animal he thought of. If it doesn't, it learns the animal.
- This game is based on an abstract type of data that is a binary decision tree.
  - Assume that the database consists of a recursive structure of type:

    (<question> <yes> <no>)

  - The initial data structure could be as follows:
    - ("Has wings" ("Flies" Ostrich Sparrow) ("Mammal" Dog Frog))
  - After the interaction it would become:
    - ("Has wings" ("Flies" Sparrow
      ("Nothing" Ostrich Penguin))
      ("Mammal" Dog Frog))

# EXERCISE 2

- Build a program for:

1. read from a file "animals.dat" an initial data structure, representing the binary tree of the game.
2. Interact with the user in the normal way for this game,
3. Update the data structure as described in the example,
4. And at the end of the game, write it in the same file.

# EXERCISE 2 (IMPLEMENTATION)

The abstract data type "binary tree" needs to be defined and implemented using a 3-element recursive "node" type.

Example of implementation:

```
(defun write-tree (no & optional (g t))
  (format g "~A" no))

(defun read-tree (&optional (f t))
  (read f))
```

```
(defun does it (yes no question)
  (list question yes no))

(defun no-query (no)
  (first no))

(defun no-yes (no)
  (second no))

(defun no-no (no)
  (third no))

(defun no-terminalp (no)
  (atom no))
```