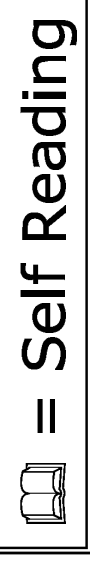




CS 236501

Introduction to AI

Tutorial 1
Introduction to Lisp

Agenda



- Historical Notes & Bibliography
- Running Lisp 
- Why Lisp?
- What Makes Lisp Different
- Lisp objects
- Inside Lisp:
 - Common Lisp Evaluation Rule & Special Operators
 - Lists: Construction, CAR and CDR
 - Assignments
 - Function Definition
 - Conditionals
 - Boolean Functions
 - Tail Recursion
 - Input – Output 

Historical Notes & Bibliography

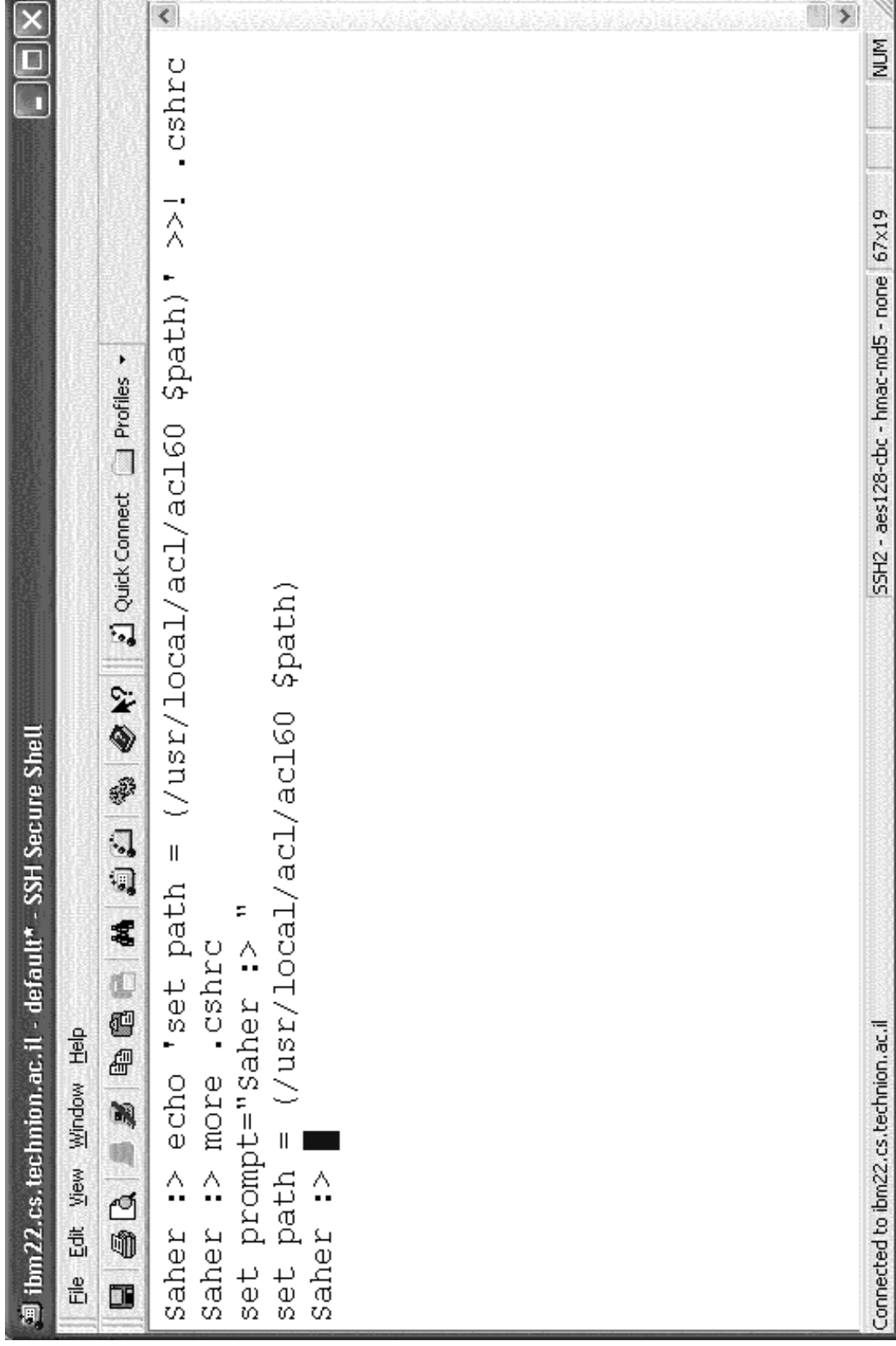
- 1958: John McCarthy and his students began work on the 1st Lisp implementation.
- 1984: The 1st Common Lisp definition.
- 1990: ANSI standard for Common Lisp.
- Bibliography:
 - ANSI Common Lisp by Paul Graham.
 - The Lisp links page in the course website (chose Links from the left menu, then Lisp).



Running Lisp I

- Detailed instructions for running Lisp could be found on the course website (chose [F.A.Q.](#) from the left menu, then [Running Lisp](#)). These include:
 - Info for installing/running [ACL](#), for which we have a new license, on windows machines (for this purpose you can borrow an installation CD from the CS library).
 - Info for running [ACL](#) on the Sun/Linux work stations.
 - Editors and a link for downloading [xemacs](#).
 - How to configure [xemacs](#) to work with [lisp](#).
 - Easy 2 steps way for running [ACL](#) from your [stud1/csd](#) account.

Running Lisp II

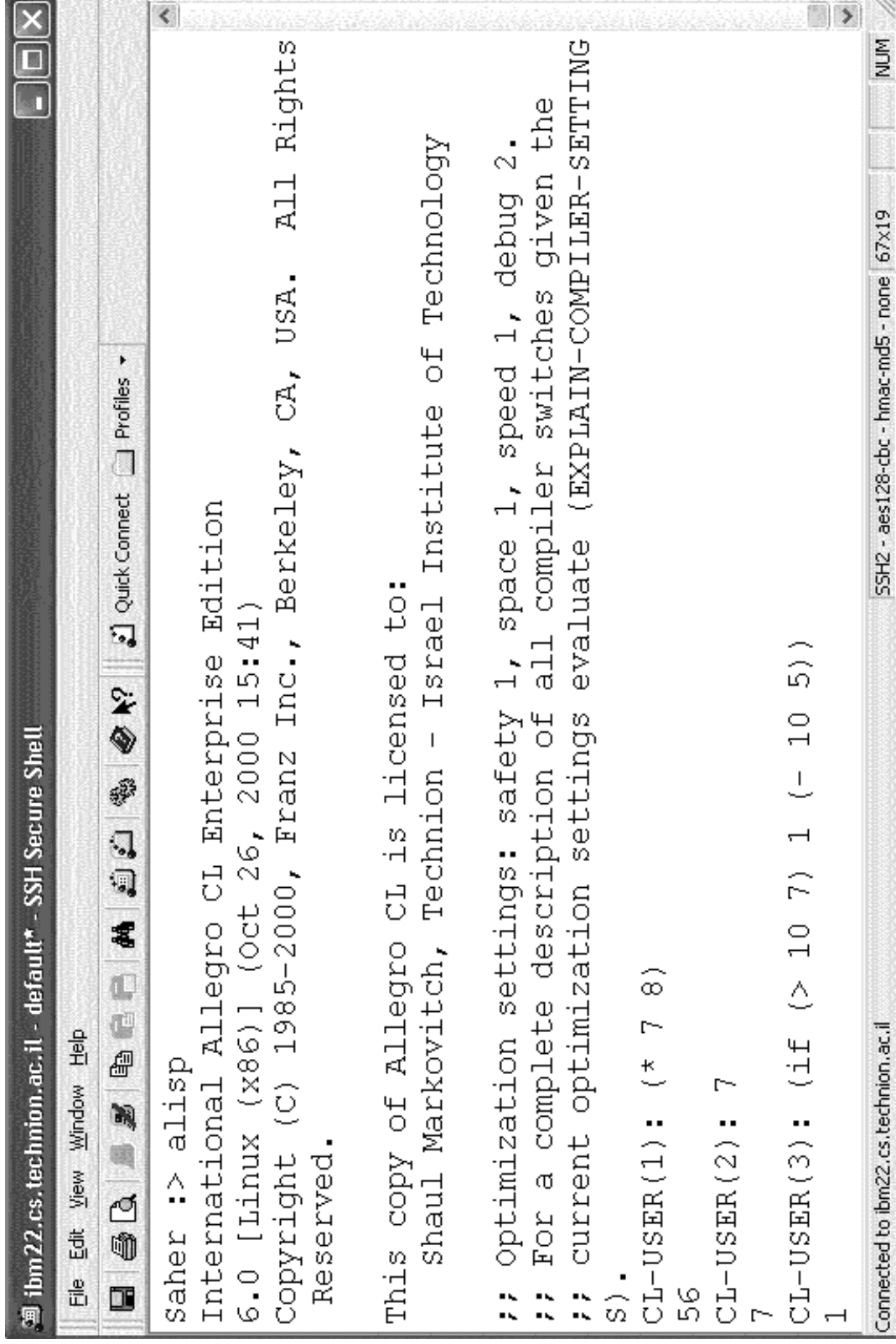


The image shows a terminal window titled "ibm22.cs.technion.ac.il - default* - SSH Secure Shell". The window has a menu bar with "File", "Edit", "View", "Window", and "Help". Below the menu bar is a toolbar with icons for file operations and a "Profiles" dropdown. The terminal content shows a user named "Saher" executing several commands in a Lisp environment:

```
Saher :> echo 'set path = (/usr/local/acl/acl60 $path)' >>! .cshrc
Saher :> more .cshrc
set prompt="Saher :> "
set path = (/usr/local/acl/acl60 $path)
Saher :>
```

The status bar at the bottom of the window displays "Connected to ibm22.cs.technion.ac.il", "SSH2 - aes128-cbc - hmac-md5 - none", and "67x19".

Running Lisp III



```
ibm22.cs.technion.ac.il - default* - SSH Secure Shell
File Edit View Window Help
Saher :> alisip
International Allegro CL Enterprise Edition
6.0 [Linux (x86)] (Oct 26, 2000 15:41)
Copyright (C) 1985-2000, Franz Inc., Berkeley, CA, USA. All Rights
Reserved.

This copy of Allegro CL is licensed to:
  Shaul Markovitch, Technion - Israel Institute of Technology

;; Optimization settings: safety 1, space 1, speed 1, debug 2.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (EXPLAIN-COMPILER-SETTING
S).
CL-USER(1): (* 7 8)
56
CL-USER(2): 7
7
CL-USER(3): (if (> 10 7) 1 (- 10 5))
1
Connected to ibm22.cs.technion.ac.il
SSH2 - aes128-cbc - hmac-md5 - none 67x19
NUM
```

Why Lisp? I

- The most popular language for AI programming.
- Functional programming paradigm.
- Let's you do things that you can't do in other languages.
 - Suppose you want to write a function that takes a number n , and returns a function that adds n to its argument. In C? In Lisp, it is as easy as:

```
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

- When does one want to do things like this?
 - Ask a Basic programmer about recursion...

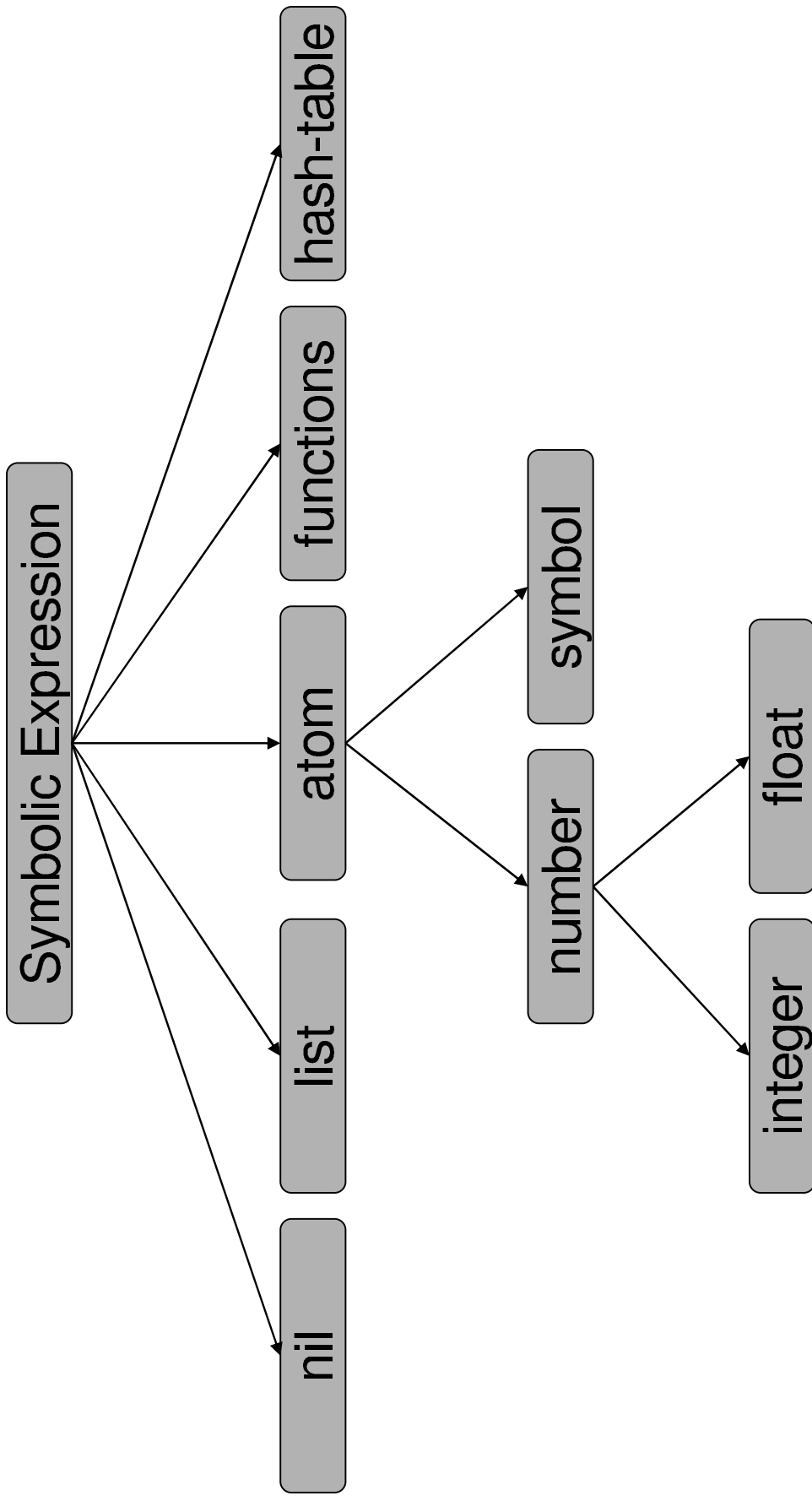
Why Lisp? II

- Allows you to write programs faster.
- The interactive nature makes it easy to experiment a program while it is being developed.
- Lisp is designed to be extendable: it lets you define new operators yourself.
 - Made of the same functions and macros as your own programs.
- Lisp is far from being an esoteric toy of CS:
 - Emacs
 - Web servers
 - NAVEX
 - Space stations experts systems

What Makes Lisp Different?

- Built-in support for lists
- Automatic storage management
 - Garbage collection
- Supports both dynamic & static typing
 - No need for type declarations. However, you may want to make them for reasons of efficiency.
- First class functions
 - Functions are regular objects like symbols, strings or lists.
 - Can be stored in variables
 - Can be passed as argument to a function
 - Can be refereed literally
- Uniform syntax

Lisp Objects



Lists

- The name “Lisp” originally stood for “List Processor”.
- Lists are one of the fundamental data structures in Lisp.
- Lists may contain any type of symbolic objects.

`(name arg1 arg2 ... argn)`

- Lists are evaluated as:
 - Functions: $arg_1, arg_2 \dots arg_n$ are evaluated and the **name** is applied.
 - Special forms: the argument evaluation policy depends on **name**.

Evaluation Rule

- Functions are always evaluated by the Common Lisp evaluation rule:
 1. The arguments are evaluated from left to right.
 2. The values of the arguments are passed to the function named `name`.
 3. If any of the arguments are themselves function calls, they are evaluated according to the same rule.
- However, not all the operators in common lisp are functions. There is a limited number of special operators, such as:
 - `if`
 - `quote`

quote

- One operator that does not follow the Common Lisp evaluation rule is `quote`.
- The evaluation rule of `quote` is: do nothing.
- The `quote` operator takes a single argument and just returns it verbatim:

```
> (quote (+ 3 5))  
(+ 3 5)  
> (quote (a b))  
(a b)
```

- For convenience, `'` is defined as an abbreviation:

```
> '(+ 3 5)  
(+ 3 5)
```

List Construction - cons

- The function `cons` builds lists.
- It takes an object and a list and returns a new list whose first element is object (1st argument) and tail is the list (2nd argument).
- Examples:

```
(cons object list)
```

```
> (cons 'a '(b c d))  
(A B C D)  
> (cons 'a (cons 'b nil))  
(A B)
```

List Construction - list

- The function `list` is a more convenient way of consing several objects onto `nil`.

```
(list object1 object2 ... objectn)
```

- It returns a new list whose elements are `object1`, `object2` ... `objectn`.
- Examples:

```
> (list 'a 'b)
(A B)
> (list '(+ 1 2) (+ 1 2))
((+ 1 2) 3)
> ' (list (+ 1 2) (+ 1 2))
(LIST (+ 1 2) (+ 1 2))
```

List Construction - append

- The function `append` concatenates all lists supplied as arguments.

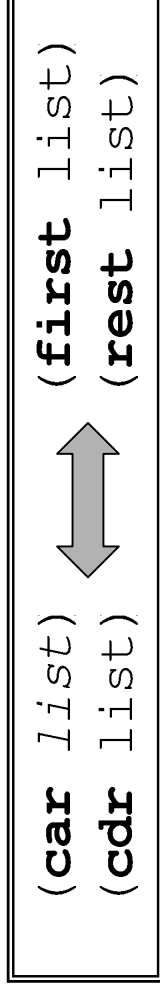
```
(append list1 list2 ... listn)
```

- It returns a new list whose elements are the elements of `list1`, `list2` ... `listn` in order. In doing so, it copies all the arguments except the last.
- Examples:

```
> (append ' (a b) ' (c d) ' (e) )  
(A B C D E)  
> (append ' (a) )  
(A)  
> (append)  
nil
```

Accessing Lists I

- The `car` (also `first`) of a list is the first element, and the `cdr` (also `rest`) is everything after the first element.



- Examples:

```
> (car ' (Intro to AI))  
INTRO ; atom  
> (first ' ((A B) c))  
(A B) ; list  
> (car ())  
NIL ; convention
```

```
> (cdr ' (Intro to AI))  
(to AI) ; list  
> (cdr (car ' ((A B) c)))  
(B) ; list  
> (rest nil)  
NIL ; convention
```

Accessing Lists II

- To find the element at a given position in a list we call `nth` (zero indexed, i.e., starts from zero):

```
(nth n list)
```

- Common Lisp defines `first` through `tenth` as functions that retrieve the corresponding element of a list (not zero indexed):

```
(third list)
```

- Examples:

```
> (nth 2 '(1 2 3))  
3
```

```
> (second '(1 2 3))  
2
```

Accessing Lists III

- To find the `nth` `cdr` we call `nthcdr`:
- Equivalent to calling `cdr` `n` times in successive.
- Examples:

```
(nthcdr n list)
```

```
> (nthcdr 1 '(1 2 3))
(2 3)
> (nthcdr 0 '(1 2 3))
(1 2 3)
> (nthcdr 7 '(1 2 3))
NIL
> (nthcdr 1 'a)
Error: Attempt to...
```

Assignments I

- `setq` and `setf` are used for assigning values to symbols (or locations).

```
(setq symbol1 value1 ... symboln valuen)  
(setf place1 value1 ... placen valuen)
```

- `setq` gives each variable `symboli` the value of the corresponding expression `valuei`. `setf` is a macro that generalizes `setq`. It stores `valuei` in the location associated with `placei`, where a valid `place` may be a variable, a call to a “settable” function, etc.... Both return the value of the last `value`.

Assignments II

- Examples:

```
> (setf x 7 y 9)
9
> x
7
> (setq y '(1 2 3))
(1 2 3)
> y
(1 2 3)
> (setq (first y) 0)
Error
> (setf (first y) 0)
0
> y
(0 2 3)
```

Function Definition

- You can define new functions with `defun`. It takes three or more arguments: a name, a list of parameters, and one or more expressions that will, make up the body of the function.

```
(defun name (param1 param2 ... paramn)  
  function-body)
```

- Examples:

```
> (defun first-and-second (l)  
  (list (first l) (second l)))  
> (setf ai-list ' (Intro to AI))  
INTRO TO AI  
> (first-and-second ai-list)  
(INTRO TO)
```

Conditionals I

- The simplest conditional in common lisp is `if`.

```
(if test then [else])
```

- Evaluates the `test` expression; If it returns `true` (`t`, `non-nil` value) the `then` expression is evaluated and its value is returned; otherwise the value of the `else` expression is evaluated and returned. In case there is no `else` expression, `nil` is returned.
- Examples:

```
> (setf x 11)
11
> (if (> x 10) (- x 10) x)
1
> (if (> x 100) x)
NIL
```

Conditionals II

- The `when` macro takes a `test` expression and a body of code.

```
(when test expression*)
```

- The `test` is evaluated. If it returns true, the body (`expressionS`) are evaluated and the value of the last `expression` is returned.
- The opposite of `when` is `unless`; it takes the same arguments, but the body will be evaluated only if the test expression returns false.

```
(unless test expression*)
```

Conditionals III

- Examples:

```
> (setf lst '(a b c))
(A B C)
> (when (listp lst) (setf is-list t) (car lst))
A
> is-list
T
> (unless (listp (car lst)) 7)
7
> (unless (listp (cdr lst)) 7)
NIL
```

Conditionals IV

- `cond` is a more versatile conditional.

```
(cond ((clause1) ... (clausen)))  
clause = test1 expression11 ... expressionin
```

- Evaluates tests until one returns `true` (`t`, `non-nil`). Only the expressions corresponding to this test are evaluated. The last evaluated expression is returned.
- Since a clause with a test `t` always evaluates to `true`, it is conventional to make the final, default clause have `t` as the test.
- If no clause succeeds the `cond` returns `nil`.

Conditionals V

- Examples:

```
> (setf x '())  
NIL  
> (cond ((null x) (setf x '(a b))) 'x-was-nil)  
(t 'x-was-not-nil)  
X-WAS-NIL  
> X  
(A B)  
> (setf temp 27)  
27  
> (cond ((> temp 35) 'too-hot)  
        ((< temp 0) 'too-cold)  
        (t 'ok))  
OK
```

Boolean Functions I

- The logical operators `and` and `or` resemble conditionals.

```
(and expression1 ... expressionn)  
(or expression1 ... expressionn)
```

- The macro `and` evaluates each expression from left to right until one of them evaluates to `nil` or non of them are left. Returns the last evaluated value.
- Example:

```
> (and 1 (> 2 1) (setf x 7) 'last)  
LAST  
> (and 1 (= 2 1) (setf x 9) 'last)  
NIL
```

Boolean Functions II

- The logical operators `and` and `or` resemble conditionals.

```
(and expression1 ... expressionn)
```

- The macro `and` evaluates each `expression` from left to right until one of them evaluates to `nil` or non of them are left. Returns the last evaluated value.
- Example:

```
> (and 1 (> 2 1) (setf x 7) 'last)
LAST
> (and 1 (= 2 1) (setf x 9) 'last)
NIL
```

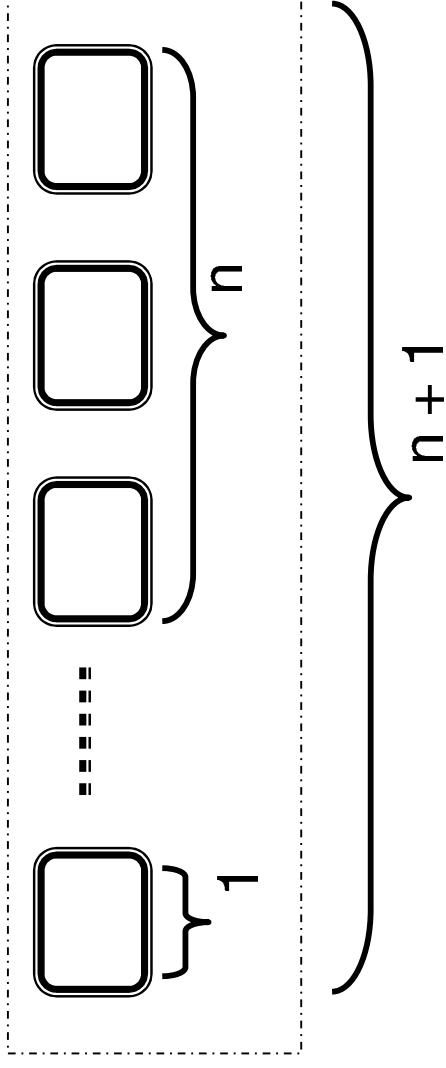
Boolean Functions III

- The macro `or` evaluates each `expression` from left to right until one of them evaluates to `true` (`t`, `non-nil`) or non of them are left. Returns `nil` when all arguments evaluate to `nil`; otherwise returns the first argument evaluating to `non-nil`.
- The `not` function just turns `non-nil` to `nil` and `nil` to `t`.
- Example:

```
> (or (> 2 1) (setf x 7))
T
> (or (not 0) (not t) (not `dog))
NIL
```

Tail Recursion I

- Consider the function below for list length:



```
(defun length1 (list)
  (if (null list) 0
      (+ 1 (length1 (rest list)))))
```

- What could be improved?

Tail Recursion II

- Tail recursion version:

```
(defun length2 (list)
  (length2-aux list 0))

(defun length2-aux (sublist leng)
  (if (null sublist) leng
      (length2-aux (rest sublist)
                    (+ 1 leng))))
```

- An improved tail recursion version:

```
(defun length3 (list &optional (leng 0))
  (if (null list) leng
      (length3 (rest list)
                (1+ leng))))
```

Recursion: A Quiz

```
(defun my-append (L1 L2)
  (cond ((null L1) L2)
        ((null L2) L1)
        ((or (atom L1) (atom L2))
         (error "wrong arguments"))
        (t ?)))
```

```
(cons (first L1) (my-append (rest L1) L2))
```

```
(defun count (List)
  (cond ((null List) 0)
        ((atom (first List))
         (1+ (count (rest List))))
        (t ?)))
```

```
(+ (count (first List)) (count (rest List)))
```

Input – Output (I)

- **Basic Input:**
 - `read` for reading expressions
 - `read-char` for characters
 - `read-line` for lines, i.e., strings up to end-of-line
- **Basic Output:**
 - `prin1` displays objects to be read by people
 - `princ` displays objects to be read by programs
 - `print` like `prin1` but prints a new line first
- **Streams are lisp objects representing sources and/or destinations of characters. To read from or write to a file, you can open it as a stream. The default input stream is `*standard-input*` while the default output stream is `*standard-output*`.**

Formatted Output

- `format` (similar to C's `printf`) can be used for sending nicely formatted output to a stream. It takes a stream (or `nil` or `t`) and zero or more additional arguments (format string that contains directives and objects to be printed).
 - When given `t` output is sent to `*standard-output*`.
 - When given `nil` output is returned as string. In all other cases, `format` returns `nil`.
 - Some directives:
 - `~f` for float, can control the number of printed digits.
 - `~%` for new line
 - `~a` anything (as printed by `princ`).
 - `~s` anything (as printed by `prin1`).

Output Examples

```
> (princ "hello")
hello
"hello"
> (print "hello")
"hello"
"hello"
> (format t "~a plus ~s is ~f" "two" "two" 4)
two plus "two" is 4.0 ;output
NIL ;returned value
```