

TUTORIAL 1

CSCI3230 (2013-2014 First Term)

By Paco WONG (pkwong@cse.cuhk.edu.hk)

Tutors

- Paco WONG
 - Email: pkwong@cse.cuhk.edu.hk
 - Office: SHB1013
- Antonio SZE-TO
 - Email: hyszeto@cse.cuhk.edu.hk
 - Office: SHB1013
- Qin CAO
 - Email: qcao@cse.cuhk.edu.hk
 - Office: SHB1013



Assessment

Assessment		Mark Distribution
Written Assignments	1 compulsory	5%
	2 optional	0%
Lisp Programming	Optional	0%
Prolog Programming	Compulsory	10%
Data Mining	Compulsory	10%
Neural Network Project	Compulsory	20%
Final Examination	Compulsory	55%

Plagiarism


Zero tolerance

- <http://www.cuhk.edu.hk/policy/academichonesty/>
- We DO encourage discussion of the course materials and assignments
- but please **DO NOT** write down anything during discussion with your friends.

Submission System

CSCI3230

Fundamentals of Artificial Intelligence



[Home](#)
[Lecture Notes](#)
[Tutorial Notes](#)
[Assignments](#)
[Useful Information](#)
[Course Consultation Group](#)

Academic Honesty

The Chinese University of Hong Kong places very high importance on honesty in academic work submitted by students, and adopts a policy of **zero tolerance** on cheating in examinations and plagiarism. Any related offence will lead to disciplinary action including termination of studies at the University.

Although detected cases of cheating in examinations or plagiarism are rare at the University, everyone should make himself/herself familiar with the content of the [website](#) and thereby help avoid any practice that would not be acceptable.

Submission System

[Login to Submission System](#)

Outline

- Introduction
- Basic concepts
 - and their evaluation
- List manipulation
- Lispbox

LISt Processing

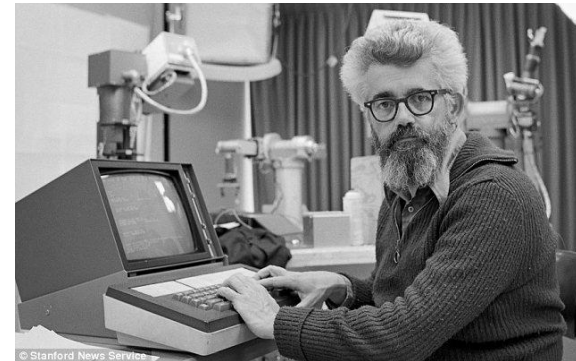
- Originally specified in 1958 (2nd oldest HL programming language)
- Designed by [John McCarthy](#)
- Many [dialects](#): Clojure, **Common LISP**, Maclisp ...

[Recursive functions of symbolic expressions and their computation by machine, Part I](#)

[J McCarthy](#) - Communications of the ACM, 1960 - dl.acm.org

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at MIT. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could ...

[Cited by 1325](#) [Related articles](#) [Library Search](#) [All 123 versions](#)



© Stanford News Service

BASIC CONCEPTS

`(setq concept (/ 1.5 1/2))`

Symbolic-Expression

- An s-expression is defined **recursively**:
 1. An atom is an s-expression
 2. If s_1, s_2, \dots, s_n are s-expressions, then so is the list $(s_1 s_2 \dots s_n)$.

Atom

- The most basic (**indivisible**) unit in LISP
- Any combination of characters, except "(" and ")", can be an atom.
- An atom with " "(whitespace) character needs to be written as
|an atom is here|
- 3 types of Atom:
 - Symbols
 - Case insensitive.
 - E.g. John, abc, 23-Jordan
 - Numbers
 - E.g. 123, 0
 - $3/4$; rational number
 - $\#C(3\ 4)$; complex number = $3 + 4j$
 - $\#C(3/4\ 4)$; complex number = $0.75 + 4j$!= $3/4 + 4j$
 - Constants (self-evaluating)
 - Symbols that have special meaning
 - E.g. NIL, T

;Comment here

List

- A nonatomic s-expression.
- A **collection** of atom or list enclosed by parentheses **()**.
- (Jordan 23) ; a list of atoms "Jordan" and "23"
- (Jordan (3/4 23)) ; a list of atoms Jordan and a list of "3/4" and "23"
- () ; **a null list**

NIL and T

- NIL
 - An special **list** called the null (empty) list ()
 - Also an **atom**
 - Means "false" or "nothing"
 - ANY non-"NIL" symbol is considered as "true" in LISP
 - A subtype of everything
- T
 - Reserved as the default symbol for "**true**".
 - ALL the data types are subtypes of T.

Form

- A form is an s-expression that is **intended** to be **evaluated**.
- If it is a list, the first element is treated as the **operator** (functions, macros or special forms) and the **subsequent elements are evaluated** to obtain the function arguments.

Example 1

(+ 2 4)

Evaluation

- If the s-expression is a number
 - Return the value of the number
- If the s-expression is an atomic symbol
 - If the symbol is bounded to a value
 - Return the value bound to that symbol
 - Otherwise
 - Error
- If the s-expression is a list
 - Evaluate the second through the last arguments
 - Apply the function indicated by the first arguments to the results.

recursion



Evaluation

(+ 2 4)

- 1) If the s-expression is a number
 - 1) Return the value of the number
- 2) If the s-expression is an atomic symbol
 - 1) If the symbol is bounded to a value
 - 1) Return the value bound to that symbol
 - 2) Otherwise
 - 1) Error
- 3) If the s-expression is a list
 - 1) Evaluate the second through the last arguments
 - 2) Apply the function indicated by the first arguments to the results.

Evaluation

- 1) If the s-expression is a number
 - 1) Return the value of the number
- 2) If the s-expression is an atomic symbol
 - 1) If the symbol is bounded to a value
 - 1) Return the value bound to that symbol
 - 2) Otherwise
 - 1) Error
- 3) If the s-expression is a list
 - 1) Evaluate the second through the last arguments
 - 2) Apply the function indicated by the first arguments to the results.

Example 2

- (+ 2 4)
- [1]
- →[2]
- →[3]
- →[3.1]
 - 2
 - [1]→[1.1]→Return value 2
 - 4
 - [1]→[1.1]→Return value 4
- →[3.2]
 - Return the value of 2+4=6
- 6

Evaluation

a

- 1) If the s-expression is a number
 - 1) Return the value of the number
- 2) If the s-expression is an atomic symbol
 - 1) If the symbol is bounded to a value
 - 1) Return the value bound to that symbol
 - 2) Otherwise
 - 1) Error
- 3) If the s-expression is a list
 - 1) Evaluate the second through the last arguments
 - 2) Apply the function indicated by the first arguments to the results.

Evaluation

- 1) If the s-expression is a number
 - 1) Return the value of the number
- 2) If the s-expression is an atomic symbol
 - 1) If the symbol is bounded to a value
 - 1) Return the value bound to that symbol
 - 2) Otherwise
 - 1) Error
- 3) If the s-expression is a list
 - 1) Evaluate the second through the last arguments
 - 2) Apply the function indicated by the first arguments to the results.

How about

- a
- [1]
- →[2]
- →[2.1]
 - Symbol “a” isn’t bounded to value “a”.
- →[2.2]
- →[2.2.1]
- **Error: Attempt to take the value of the unbound symbol A**

How can we differentiate atomic symbol “a” from value “a”?

Control the Flow of Evaluation

- **QUOTE** is a special form that returns its arguments **without evaluation**.
- Often, the argument is to be treated as **data** rather than an **evaluable form**.

Example 3

> (quote a)	; "a" is quoted
A	; the argument of quote is "A"
> 'a	; "a" is equivalent to "(quote a)"
A	; same result

- **EVAL** tells the interpreter to evaluate the expression

Example 4

> (eval '(* 2 3))	;
6	; eval undoes the effect of quote

Binding Variables Explicitly

- **SETF** is a macro that examines an access form and produces a call to the corresponding update function.
- It uses its first argument to obtain a **memory location** and places the value of the second argument in that location.

Example 6

```

> (setf index (+ 2 6))      ;
8                           ; symbol "index" is set to 8
> (setq a '(1 2 3))        ; create a list of '(1 2 3)
(1 2 3)
> (setf (car a) 4)          ; car is a function that returns the head of the list
(4 2 3)
> a
(4 2 3)

```

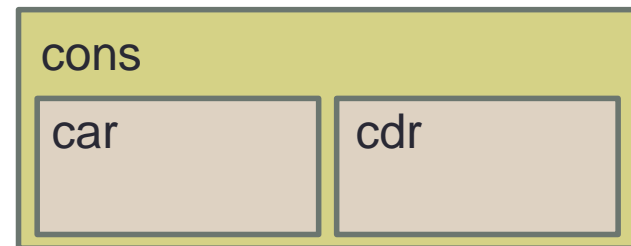
Does *setf* evaluate the first argument?

LIST MANIPULATION

```
(append (cons 'c '(s c i)) '(3 2 3 0))
```

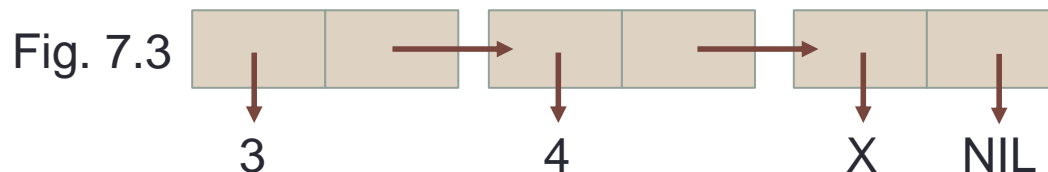
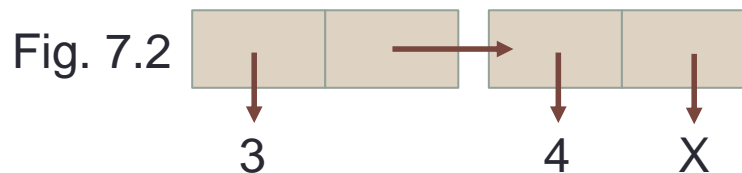
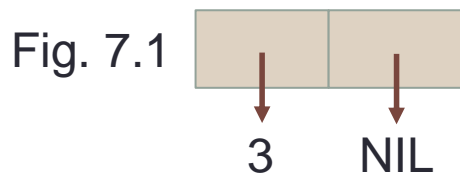
Cons cell: A two-field record

- A two-field record in memory
- The fields are called “car” and “cdr”



Example 7

> (cons 3 nil)	; fig. 7.1
(3)	; 3 is car and NIL is cdr, last nil isn't printed
> (cons 3 (cons 4 'x))	; fig. 7.2
(3 4 . X)	; dotted-pair notation, improper list
> (cons 3 (cons 4 (cons 'x nil)))	; fig. 7.3
(3 4 X)	; list notation, equivalent to (3 . (4 . (X . NIL)))



Lists: implemented on top of cons pairs

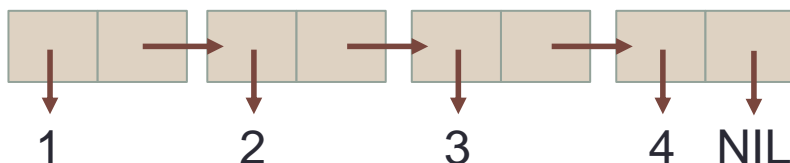
• LIST

Example 8

```
> (list 3 4 'x)      ; non-numeric atom
(3 4 X)              ; a list is created
> '(3 4 x)           ; equivalent?
(3 4 X)              ; seems to be
> (list 3 (+ 2 2) x) ; list is more powerful
(3 4 X)              ; same as above
```

Example 9

```
> (setq L (list 1 2 3 4))
(1 2 3 4)
> (car L)
1
> (cdr L)
(2 3 4)
```



• PUSH and POP

Example 10

```
> (setq a nil)
NIL
> (push 3 a)
(3)
> (push 5 a)
(5 3) ; the leftmost is the top of the stack
        ; a is modified
> (pop a)
5
> a
(3)
> (pop a)
3
> (pop a)
NIL
> a
NIL
```

Try (push 5 '(3))

C[AD][AD]*R and NTHCDR

- ▶ The definition of a CAR and CDR of NIL is also NIL.
- ▶ C [AD][AD]* R is the combination of CAR and CDR, e.g. CDDR, CADDR.
- ▶ NTHCDR is useful if the number of CDR varies.

Example 11

```
> (car NIL)
NIL ; behave as an atom
> (cdr NIL)
NIL ; behave as a list
```

Example 12

```
> (cadr '(a b c)) ; equivalent to (car (cdr '(a b c)))
B ; get the 2nd element of the list
> (cdar '((1 . 2) 3)) ; another example
2
```

Example 13

```
> (nthcdr 0 '(1 2 3 4 5)) ; does not perform cdr
(1 2 3 4 5)
> (nthcdr 3 '(1 2 3 4 5)) ; equivalent to (cdr (cdr (cdr '(1 2 3 4 5))))
(4 5)
```

NTH, APPEND, DELETE

Many more operators!!!

- **NTH** is to get the $(n+1)$ th element of the list.
- **APPEND** **concatenates** lists into one list. It creates a new list and does NOT modify the argument lists.
- **DELETE** removes its first argument from its second. The second must be a list.

Example 14

```
> (nth 5 '(a b c d e f))
F
```

Example 15

```
> (append '(a b) 'c)
(A B . C)
```

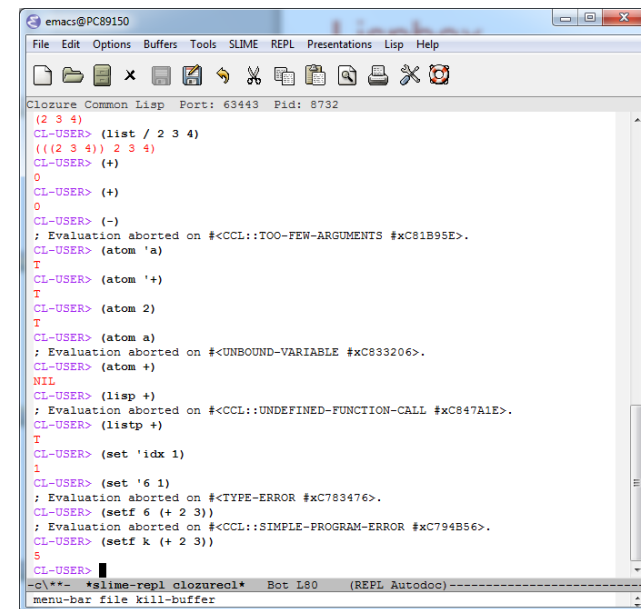
Example 16

```
> (delete 'b '(a b c))
(A C)
> (delete 'd '(a b c))
(A B C) ;no error message
> (delete 'a '(a a a b a))
(B)
```

For more operators to manipulate list, please read Chapter 15 of
<http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node1.html>

Lispbox

- An IDE for Clozure Common Lisp development.
 - All-in-one: to get you up and running in a good Lisp environment as quickly as possible (CL, Emacs with SLIME)
 - <http://common-lisp.net/project/lispbox/>
- Run your program
 - Open the .lisp file
 - Edit → Select All
 - SLIME → Evaluation → Eval Region
- When you encounter an error, press q.
- Explore the GUI for more commands



The screenshot shows the Lispbox IDE window titled "emacs@PC89150". The menu bar includes File, Edit, Options, Buffers, Tools, SLIME, REPL, Presentations, Lisp, and Help. The toolbar contains icons for file operations, editing, and evaluation. The main window displays a Clozure Common Lisp REPL session with the following text:

```

Clozure Common Lisp Port: 63443 Pid: 8732
(2 3 4)
CL-USER> (list / 2 3 4)
(((2 3 4)) 2 3 4)
CL-USER> (+)
0
CL-USER> (+)
0
CL-USER> (-)
; Evaluation aborted on #<CCL::TOO-FEW-ARGUMENTS #xC81B95E>.
CL-USER> (atom 'a)
T
CL-USER> (atom '+)
T
CL-USER> (atom 2)
T
CL-USER> (atom a)
; Evaluation aborted on #<UNBOUND-VARIABLE #xC833206>.
CL-USER> (atom +)
NIL
CL-USER> (list +)
; Evaluation aborted on #<CCL::UNDEFINED-FUNCTION-CALL #xC847A1E>.
CL-USER> (listp +)
T
CL-USER> (set 'idx 1)
1
CL-USER> (set '6 1)
; Evaluation aborted on #<TYPE-ERROR #xC783476>.
CL-USER> (setf 6 (+ 2 3))
; Evaluation aborted on #<CCL::SIMPLE-PROGRAM-ERROR #xC794B56>.
CL-USER> (setf k (+ 2 3))
5
CL-USER>

```

The status bar at the bottom shows "c:*- *slime-repl clozurecl* Bot L80 (REPL Autodoc)-----" and "menu-bar file kill-buffer".

Summary

- S-expression
 - Atom and list
- Evaluation and its control
 - Form, QUOTE and EVAL
- Binding variable explicitly
 - SET, SETQ and SETF
- Cons cell and list
 - CONS, CAR, CDR, NTHCDR, NTH, APPEND, ...
- Lispbox

Feedback: email or <http://goo.gl/5VLYA>

Suggested Readings

- Common Lisp the Language, 2nd Edition
 - <http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>
- <http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf>
- Common LISP Hints
 - <http://www.carfield.com.hk/document/languages/common-lisp-tutorial.html>

Reference

- 2008/2009 First Term Tutorial Notes by David Lam
- Common Lisp the Language, 2nd Edition
 - <http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>
- <http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf>
- Common LISP Hints
 - <http://www.carfield.com.hk/document/languages/common-lisp-tutorial.html>
- George F. Luger. Artificial Intelligence 5th ed. Addison Wesley.
- Wendy L. Milner. Common LISP, a Tutorial. Prentice Hall.

Appendix: Data types

- Data Types
 - Numbers
 - Integers
 - Ratios
 - Floating-Point Numbers
 - Complex Numbers
 - Characters
 - Standard Characters
 - Line Divisions
 - Non-standard Characters
 - Character Attributes
 - String Characters
 - Symbols
 - Lists and Conses
 - Arrays
 - Vectors
 - Strings
 - Bit-Vectors
 - Hash Tables
 - Readtables
 - Packages
 - Pathnames
 - Streams
 - Random-States
 - Structures
 - Functions
 - Unreadable Data Objects
 - Overlap, Inclusion, and Disjointness of Types

Reference:

<http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node1.html>

SETF, SET and SETQ



Originally, in Common Lisp, there were no lexical variables -- only dynamic ones. And there was no SETQ or SETF, just the SET function.

What is now written as:

```
(setf (symbol-value '*foo*) 42)
```

was written as:

```
(set (quote *foo*) 42)
```

which was eventually abbreviated to SETQ (SET Quoted):

```
(setq *foo* 42)
```

Then lexical variables happened, and SETQ came to be used for assignment to them too -- so it was no longer a simple wrapper around SET.

Later, someone invented SETF (SET Field) as a generic way of assigning values to data structures, to mirror the l-values of other languages:

```
x.car := 42;
```

would be written as

```
(setf (car x) 42)
```

For symmetry and generality, SETF also provided the functionality of SETQ. At this point it would have been correct to say that SETQ was a Low-level primitive, and SETF a high-level operation.

Then symbol macros happened. So that symbol macros could work transparently, it was realized that SETQ would have to act like SETF if the "variable" being assigned to was really a symbol macro:

```
(defvar *hidden* (cons 42 42))
(define-symbol-macro foo (car *hidden*))
```

```
foo => 42
```

```
(setq foo 13)
```

```
foo => 13
```

```
*hidden* => (13 . 42)
```

So we arrive in the present day: SET and SETQ are atrophied remains of older dialects, and will probably be booted from eventual successors of Common Lisp.

Bound and free variables in a function

- A bound variable is one that appears as a formal parameter in the definition of the function, while a free variable is one that appears in the body of the function but is not a formal parameter. When a function is called, any bindings that a bound variable may have in the global environment are saved and the variable is rebound to the calling parameter. After the function has completed execution, the original bindings are restored. Thus, setting the value of a bound variable, as seen in the LISP interaction:
- `>(defun foo(x) (setq x (+ x 1) x)`
- `foo`
- `>(setq y 1)`
- `1`
- `>(foo y)`
- `2`
- `>y`
- `1`

Symbolic programming

- Reference:
- <http://www.ai.uga.edu/mc/LispNotes/FirstLectureOnSymbolicProgramming.pdf>
- “People write formulas that say how to manipulate formulas.” That is the key idea McCarthy introduced – that we can define functions of expressions, not just functions of numbers or strings or bits or other data types.
- ... Symbolic programming is programming in which **the program can treat itself, or material like itself, as data** – programs can write programs.
- Generally, in symbolic programming languages, **complex data structures are easy to create** – you can usually create them by just writing them, and then extend them by performing simple operations. If you’ve ever created a linked list in C, you know it takes a lot of function calls to do so. Not so in Lisp or Prolog.