

CS 161 Intro. To Artificial Intelligence

Week 1, Discussion 1C

Li Cheng Lan

General Information

- **Course info:**
 - 8 Projects - 4 LISP programming assignments + 4 written assignments (tentative)
 - More details about midterm/final will come later this quarter
- **TA – Li Cheng Lan (Benson):** lclan@cs.ucla.edu
- **Discussion 1C:**
 - Fridays 2:00 pm - 3:50 pm
- **Office hours:**
 - Thur 11:30 am - 1:30 pm

Today's Topics:

- Background of Lisp
- Running Environments of Lisp
- Lisp Syntax
- Example Functions



Background of Lisp

- **LISP** derived from “**LISt Processor**”
 - Use linked-list as one of major data structures
 - Source code are made of lists
 - Many dialects (Common Lisp, Scheme, etc.)
- Appeared in 1958 (2nd oldest high-level programming language)
- In this class, we mainly use **Common Lisp** -- a general-purpose, multi-paradigm programming language.
 - Object-oriented and fast prototyping capability

Running Environments

- CLISP: implements the language described in the ANSI Common Lisp standard with many extensions.
 - <https://clisp.sourceforge.io/>
 - You can also access it from SEASnet
- Lisp online: <https://jscl-project.github.io/>
 - Good for testing syntax and single functions
- Tutorials:
 - <https://www.tutorialspoint.com/lisp/>
 - Practical Common Lisp (Book) -- free online!
<http://www.gigamonkeys.com/book/>

LISP Online

← → ↻ 🏠 🔒 jscl-project.github.io

Welcome to JSCL 0.7.0 (built on 27 November 2018)

JSCL is a Common Lisp implementation on Javascript.
For more information, visit the project page at [GitHub](#).

CL-USER> (+ 2 4)

6

CL-USER> (/ 6 4)

1.5

CL-USER> (cons 'A '(B))

(A B)

CL-USER> (defun AddOne (N) (+ 1 N))

...)

...

... (defun MinusOne (N) (- N 1))

...)

ADDONE

CL-USER> (addone 2)

3

CL-USER> (minusone 2)

ERROR: Function 'MINUSONE' undefined

CL-USER> █

CLISP on SEASnet

Requirements:

- SEASnet account
 - <https://www.seasnet.ucla.edu/seasnet-accounts/>
- Connection to SEASnet server (**important**)
 - Use Cisco VPN connect to campus network!
 - <https://www.it.ucla.edu/it-support-center/services/virtual-private-network-vpn-clients>
- Some familiarity to Linux commands

CLISP on SEASnet

- Log into your SEASnet account:
 - macOS -> terminal: `ssh -X lnxsrv.seas.ucla.edu -l yourseasaccountname`
 - Windows -> putty: SSH to [yourseasaccountname@lnxsrv.seas.ucla.edu](#)
 - Download putty: <https://www.chiark.greenend.org.uk/~sgtatham/putty/>
- Copy Lisp file to SEASnet:
 - `scp -r LocalPath/file yourSEASaccount@lnxsrv.seas.ucla.edu:./SEASpath`
- **Interactive mode:**
 - `clisp`
 - *(quit)* or *(exit)* to leave
- **Load from file:**
 - `clisp ./SEASpath/file`

```
[xiao@lnxsrv03 ~/Documents/cs161]$ clisp
      i i i i i i i      oooooo  o      oooooo  oooooo
      I I I I I I I      8      8      8      8      8      8
      I \ '+' / I      8      8      8      8      8      8
      \ '-+-' /      8      8      8      oooooo 8ooooo
      '-_+_'      8      8      8      8      8      8
      |      8      o      8      8      o      8      8
      -----+-----      oooooo 8ooooooo 8oo8ooo  oooooo  8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010
```


Lisp Syntax

- Two fundamental pieces:
 - Atom
 - Symbolic expression (S-expression)
- Not case sensitive
- Use “;” to comment

Lisp Syntax

Atom: all objects except cons cells (lists), can't be further divided

- 3 ; => 3
- '/' ; single character symbol
- "Hello, world!" ; all of the quoted text is a single atom
- t ; boolean True. In Lisp **any non-nil value is True!**
- nil ; boolean False **OR** an empty list ()
- A ; Error! Not defined.

Lisp Syntax

S-expression (symbolic expression): simple, no operator priority, no ambiguity

- (function arg1 arg2 ... argN)
 - Eg. (+ 3 5) ; => 8
- (+ 2(/ (* 2 10) 5)) ; represents $2 + 2 * 10 / 5$
- Use *quote* or ' to prevent it from being evaluated:
 - '(+ 3 5) ; => (+ 3 5)
 - (quote (* 2 4)) ; => (* 2 4)
 - '(1 2 3) ; list (1 2 3)

Lisp Syntax

Basic data types:

- Numeric:
 - Integers (e.g. 2), floating point (e.g. 2.0), ratios (e.g. $\frac{1}{2}$)
 - Binary (e.g. #b111, output is 7), hexadecimal (#x111, which is 273)
 - Complex numbers (e.g. #C(1 2))
- Symbolic:
 - It's a name that represents data objects and it's also a data object.
 - It contains a property list, or plist.
 - LISP allows you to assign properties to symbols (e.g. 'A)
- Boolean:
 - False (nil) and True (t or any non-nil value, e.g. 0)

Lisp Syntax

Basic arithmetic operations:

- `(+ 2 3)` ; => 5
- `(- 5 1)` ; => 4
- `(* 10 2)` ; => 20
- `(expt 2 3)` ; => 8
- `(mod 7 2)` ; => 1
- `(/ 20 4)` ; => 5
- `(/ 2 3)` ; => 2/3 or 0.6666...

Lisp Syntax

Booleans and Equity:

- **(not nil)**
=> t
 - **(and 0 t)**
=> t
 - **(and 0 1 2)** ; => 2
 - **(or 0 nil)**
=> 0
 - **(or 0 1 2)**
=> 0
 - **(and 1 ())**
- Compare numbers using “=”
; ○ **(= 2 2.0)** ; => t
 - Compare object identity using “eql”
; ○ **(eql 2 2)** ; => t
○ **(eql 2 2.0)** ; =>
nil
; ○ **(eql 'A 'A)** ; => t
○ **(eql (list 3) (list 3))** ; => nil
; Note: eql compares based on memory chunk
 - Compare lists, strings using “equal”
; ○ **(equal (list 'A) (list 'A))** ; => t

Lisp Syntax

Strings:

- Concatenation of two strings using “concatenate”:
 - `(concatenate 'string "aa" "bb")` ; => “aabb”
- Print a string using “print”:
 - `(print “hello”)` ; prints “hello” and returns “hello”
 - `(+ 1 (print 2))` ; prints 2, returns 3.
- Return/print a string using “format”:
 - `(format nil “hello ~a” “alice”)` ; return “hello alice”
 - `(format t “hello ~a” “alice”)` ; prints “hello alice”, return nil
 - “~a” for string, “~d” for integer, “~2f” for float

```
(concatenate 'string "aa" "bb")
```

```
[48]> (print "hello")
"hello"
"hello"
[49]> (+ 1 (print 2))
2
3
```

Lisp Syntax

Variables:

- Global variables:
 - Variable name can contain any character except `()",`";#\`
 - `(defparameter age 24)` ; define a variable age with value 24
 - `(defparameter *age* 30)` ; define a variable *age* (not age) with value 30
 - `(defparameter age 30)` ; age => 30, value of variable changed!
 - `(defvar *city* "LA")` ; *city* => "LA"
 - `(defvar *city* "NYC")` ; *city* => "LA", defvar doesn't change value!

Lisp Syntax

Variables:

- Local variables:
 - Use “let” or “let*” statement
 - “let” does parallel assignment, “let*” does sequential assignment
 - (**let** ((var1 value1) (var2 value2) ... (varN valueN)) (s-expression))

E.g. (let ((a 10) (b 20))

(+ a b)

)

Binding

Body

; returns 30

- For homeworks, you are **NOT** allowed to use global variables. You can only use let/let*.

Lisp Syntax

Lists: uses linked-list data structure, made of atoms and/or CONS pairs

- Construct lists using “cons” (take **only two** arguments):
 - (cons 2 '(B)) ;=> '(2 B)
 - (cons 2 'B '(C)) ;=> Error! Too many arguments.
 - (cons 1 (cons 2 (cons 3 nil))) ;=> '(1 2 3)
 - (cons '(A C) '(9 4)) ;=> '((A C) 9 4)
- Construct lists using “list” (take **multiple** arguments):
 - (list 1 2 '3) ;=> '(1 2 3)
 - (list '(A (C)) '(9)) ;=> '((A (C)) (9))
 - (list 1 2 nil) ;=> (1 2 nil)

Lisp Syntax

Lists:

```
Break 11 [12]> (defun same (x) (cons (car x) (cdr x)))  
SAME  
Break 11 [12]> (same '(1 2 3))  
(1 2 3)  
Break 11 [12]>
```

- Construct lists using “append” (**only take lists** as args, can have **multiple args**):
 - (**append** 2 'B) ; Error! 2 and 'B are not lists
 - (append '(2) '(B) '(C)) ; => (2 B C)
 - (append '(A) '(F (2)) nil '(H)) ; => (A F (2) H)
- Parse lists using “car”/“first”, “cdr”/“rest”, “cadr”, “caddr”, etc. (only works for list):
 - (**car** '(1 2 3 4)) or (**first** '(1 2 3 4)) ; => 1
 - (**cdr** '(1 2 3 4)) or (**rest** '(1 2 3 4)) ; => (2 3 4), cdr always return a list
 - (**caddr** '(1 2 3 4)) ; => 3, caddr operates from right to left
 - “cons” is the reverse of “car” + “cdr”

Lisp Syntax

Functions:

- Define a function using “defun”:
 - (**defun** functionName (arg1 ... argN) (s-expression))
 - E.g. (defun sayHello (name) (format nil "Hello, ~A" name))
- Call a function by its name:
 - (sayHello “Sam”) ; => “Hello, Sam”

Lisp Syntax

Control Flow:

- If-statement:

- **(if (test expression) (then expression) (else expression))**

- E.g. (if (equal name "Fred") ; test expression

"Found Fred!" ; then expression

"Not found.") ; else expression

- Chain of tests using "cond":

- **(cond (cond1 value1) (cond2 value2) ... (t valueT))**

- E.g. (cond ((> *age* 20) "Older than 20")

((< *age* 20) "Younger than 20")

(t "Exactly 20")) ; without default case, it returns nil if *age*==20

Default case

You are encouraged to use cond (instead of if) in homeworks

Lisp Syntax

Control Flow:

- Recursion: “cond” becomes very useful
 - E.g. Check if a list x contains an element e

```
(defun contains (e x)
  (cond ((not x) nil) ; base case when x is empty
        ((atom x) (equal e x)) ; base case when x has only one item
        (t (or (contains e (car x)) (contains e (cdr x))))) ; recursive step
  )
)

(contains 'c '(a (2 d) c e)) ; => t
```

Lisp Syntax

Control Flow:

- Iteration:
 - E.g. Print out numbers from 1 to 5
(loop for x in '(1 2 3 4 5)
do (print x))
- For homeworks, you are **NOT** allowed to use iteration, you can **only use recursion**.

prints: 1

2

3

4

5

return: nil

```
Break 11 [12]> (loop for x in '(1 2 3 4 5)
do (print x) )

1
2
3
4
5
NIL
```

Example Functions - Using Recursion

- Factorial
- Compute list length
 - Top-level
 - Deep-level
- Check if a list contain a specific element
- Check if a list contain any number
- Find k-th element (top-level)
- Delete k-th element (top-level)

Example Functions

Factorial: (factorial 5) => 120

Example Functions

Factorial:

```
(defun factorial (n)
```

```
  (if (< n 2)
```

```
      1
```

; when $n < 2$ return 1

```
      (* n (factorial (- n 1)))))
```

; when $n \geq 2$ do recursive steps

```
  )
```

```
)
```

```
(factorial 5)
```

; => 120

Example Functions

Compute list length (top-level):

E.g. Top-level length of `'((a b) c ((1 2) d))` is 3.

Example Functions

Compute list length (top-level):

E.g. Top-level length of '((a b) c ((1 2) d))' is 3.

```
(defun listlength (x)
  (if (not x)
      0 ; when x is empty return 0
      (+ (listlength (cdr x)) 1) ; otherwise do recursive steps
  )
)
```

Example Functions

Compute list length (deep-level):

E.g. Deep-level length of `'((a b) c ((1 2) d))` is 6.

Example Functions

Compute list length (deep-level):

E.g. Deep-level length of '((a b) c ((1 2) d))' is 6.

```
(defun deeplength (x)
  (cond ((not x) 0) ; empty list, returns 0
        ((atom x) 1) ; atom: check if x is an atom. If yes returns 1
        (t (+ (deeplength (car x)) (deeplength (cdr x))) ; else, do
              recursive steps
            )
        )
  )
)
```

Example Functions

Check if a list contain a specific element:

(contains '2 '(4 3 (1 2) 8))

;=> t

Example Functions

Check if a list contain a specific element:

```
(defun contains (e x)
  (cond ((not x) nil) ; base case when x is empty
        ((atom x) (equal e x)) ; base case when x has only one item
        (t (or (contains e (car x)) (contains e (cdr x)))) ; recursive step
  )
)
```

```
(contains '2 '(4 3 (1 2) 8)) ; => t
```


Example Functions

Check if a list contain any number:

`(contains_number '(a b c))` ; => nil

`(contains_number '(a (b 2) c))` ; => t

Example Functions

Check if a list contain any number:

```
(defun contains_number (x)
  (if (atom x)                                ; nil if x is a list
      (numberp x)                             ; numberp: check if x is a number
      (or (contains_number (car x)) (contains_number (cdr x))) ; recursively flatten
  )
)
```

```
(contains_number '(a b c))                    ; => nil
```

```
(contains_number '(a (b 2) c))                ; => t
```

Example Functions

Find k-th element (top-level):

```
(find_kth 4 '(a (b c) d (e f) g)) => (e f)
```

Example Functions

Find k-th element (top-level):

```
(defun find_kth (k x)
  (if (= k 1)
      (car x) ; if k is 1 return the 1st item
      (find_kth (- k 1) (cdr x)); else pass (k-1) and rest of list to find_kth recursively
  )
)
```

```
(find_kth 4 '(a (b c) d (e f) g)) ; => (e f)
```

Q: How do we find k-th element in deep-level? E.g. (find_kth 4 '(a (b c) d (e f) g)) => d

Example Functions

Delete k-th element (top-level):

```
(delete_kth 4 '(a (b c) d (e f) g))
```

```
;=> (a (b c) d g)
```

Example Functions

Delete k-th element (top-level):

```
(defun delete_kth (k x)
  (if (= k 1)                ; nil if k is larger than 1
      (cdr x)                 ; if k is one return the rest of list (discard the 1st item)
      (cons (car x) (delete_kth (- k 1) (cdr x))) ; else, recursive steps
  )
)
```

```
(delete_kth 4 '(a (b c) d (e f) g)) ; => (a (b c) d g)
```

Q: How do we delete k-th element in deep-level?

Questions?

- My slides take the following materials as references:
 - Shirley Chen's slides
 - Yewen Wang's slides

Thank you!