An Introduction to LISP

- ✔ How to interact with LISP:

```
sdcc5.ucsd.edu% cl

Allegro CL 4.1 [SPARC; R1] (4/9/93 17:00)
Copyright (C) 1985-1992 Franz Inc., Berkeley CA, USA.
;; Optimization settings; safety 1, space 1, speed 1, debug 2
;; For a complete description of all compiler switches
;; evaluate (EXPLAIN-COMPILER-SETTINGS).
USER(1):
```

- ✔ `cl` stands for Common LISP, now the standard dialect of LISP

- ✔ (Note the semicolons `;` In LISP programs, comments start with a semicolon and go to the end of the line)

Interacting with LISP

- LISP is an interactive language
  - (Common LISP also has a good compiler, and can do offline computations)

- When you invoke LISP, the lisp interpreter enters a `read - eval - print` loop

- You type an expression in response to the prompt, and ...
  - LISP reads in the expression you typed
  - LISP evaluates it
  - LISP prints the result of the evaluation, and prompts for the next input

- (Note: in fact, `read, eval, print` are names of builtin LISP functions...)

Interacting with LISP

- What is the result of LISP's evaluation process?
- It depends on the expression that is being evaluated
- Expressions are of two types:
  - atoms
  - lists
- Atoms are of two types:
  - constants:
    - numbers: `27, 3.14, 5/8, 1.0e26,` etc.
    - strings: `"A string"`, `"another one"`, etc.
    - characters: `#\a, #\A, #\b`, etc.
  - symbols: `foobar, FooBaR, hyphenated-symbol, +,` etc.
- A list is represented as

  (

  < zero or more atoms or lists >

)

  - examples of lists:

```
()
(item1 item2 item3)
(foo (bar baz))
(+ 3 4)
(() (()()))
```

Expression types in LISP

- s-expressions
  - atom
    - symbol
    - constant
      - number
      - string
      - character
  - list
    - the empty list
    - cons

LISP's evaluation process

- How does LISP evaluate atoms?
  - constants evaluate to themselves:

```
USER(1): 3.14
3.14

USER(2): 5/8
5/8

USER(3): 1e5
100000.0

USER(4): "this is a string"
"this is a string"

USER(5): #\a
#\a
```

  - the special symbols `t` and `nil` act like constants and evaluate to themselves:

```
USER(4): t
T
USER(5): nil
NIL
```

LISP's evaluation process

- in general, a symbol evaluates to its value, but...

```
USER(6): x
Error: Attempt to take the value of the unbound variable 'X'.
```

```
[1] USER(7): :reset
USER(8):
```

- How can we give a symbol a value without using assignment statements?...

LISP's evaluation process

- How does LISP evaluate lists?
  - The empty list evaluates to `nil`:

```
USER(10): ()
NIL
USER(11):
```

  - If the list is not empty, and the first element of the list names a function, then
    - the remaining elements of the list are evaluated,
    - and the function is called with those values as arguments

```
USER(12): (+ 2 3)
5
USER(13): (+ 2 3 4 77 5)
91
USER(14): (* 7 11)
77
USER(15): (+ 2 3 4 (* 7 11) 5)
91
```

  - The first element of the list can also name a *macro* or a *special form* (more about these later on!).

LISP's evaluation process

- ✗ If the first element of a list to be evaluated does not name a function or macro or special form, it is an error:

```
USER(16): (plus 2 3)
Error: attempt to call `PLUS' which is an undefined
function. [condition type: UNDEFINED-FUNCTION]
Restart actions (select using :continue):
0: Try calling PLUS again
1: Return a value
2: Try calling a different function
3: Setf the symbol function of PLUS and call it again

[1] USER(17): :reset
USER(18):
```

- ✗ How can we make a symbol name a function?...

Stopping evaluation

- ✔ You can prevent the evaluation of an expression by preceding it with a single-quote ’

```
USER(53): (list 3 4 5 6)
(3 4 5 6)

USER(54): (first (list 3 4 5 6))
3

USER(55): (first (3 4 5 6))

Error: Funcall of 3 which is a non-function.

USER(56): (first ’(3 4 5 6))
3

USER(57): ’x
X


USER(58): (list t nil “a string” ’(+ 2 3) (+ 2 3) ’ucsd)

(T NIL “a string” (+ 2 3) 5 UCSD)
```

List processing functions in LISP

- ✔ `first` and `rest` return parts of lists
  - ✗ `first` returns the first element of a list

```
USER(60): (first '(fast computers are nice))
FAST

USER(61): (first '((3 (4 5)) 6 ((7))))

(3 (4 5))
```

  - ✗ `rest` returns a list containing all but the first element of a list

```
USER(62): (rest '(fast computers are nice))
(computers are nice)

USER(63): (rest `((3 (4 5)) 6 ((7))))

(6 ((7)))
```

- ✔ `car` and `cdr` are synonyms for `first` and `rest` (for historical reasons)
- ✔ the functions `cxxr, cxxxr, cxxxxr` also exist, where `x=a` or `d`
  - ✗ `(cadr L)` is equivalent to `(car (cdr L))`
  - ✗ `(cddar L)` is equivalent to `(cdr (cdr (car L)))`
  - ✗ etc.!

List processing functions in LISP

- `nth` returns the nth element of a list, where the `car` of the list is the zeroth element

```
USER(70): (nth 0 '(foo bar gack))
FOO
USER(71): (nth 1 '(foo bar gack))
BAR
```

- `nthcdr` returns the result of applying cdr to a list n times

```
USER(72): (nthcdr 0 '(foo bar gack))
(FOO BAR GACK)

USER(73): (nthcdr 3 '(foo bar gack))


NIL
```

- note that by convention, the `car` and the `cdr` of `nil` are both `nil`

```
USER(74) (nth 99 nil)
NIL
USER(75) (cddar nil)
NIL
```

List processing functions in LISP

- ✔ `cons`, `list`, and `append` construct lists
  - ✗ `cons` takes two arguments and returns a list such that
    - the value of the first argument is the `first` of the new list
    - the value of the second argument is the `rest` of the new list

```
USER(1): (cons 1 '(2 3))
(1 2 3)

USER(2): (cons 'hello nil)
(HELLO)

USER(3): (cons (first '(what a nice list))
               (rest '(what a nice list)))


(WHAT A NICE LIST)

USER(4): (cons (first nil) (rest nil))


(NIL)
```

  - ✗ list takes any number of arguments and returns a list of their values

```
USER(5): (list '+ 1 2 3)
(+ 1 2 3)

USER(6): (list (first '(what a nice list))
               (rest '(what a nice list)))

(WHAT (A NICE LIST))
```

List processing functions in LISP

- append takes any number of arguments
  - each argument must have a list as value
  - append returns a list whose elements are the elements of the argument lists

```
USER(1): (append '(a b c) '(d e))
(A B C D E)

USER(2): (append '((a b) c) '() '(d ((e))))
((A B) C D ((E)))
```

```
USER(3): (cons '(a b c) '() )
```

```
USER(4): (list '(a b c) '() )
```

```
USER(5): (append '(a b c) '() )
```

List processing in LISP

- `reverse` takes a list as argument, and returns a list whose elements are the elements of the argument list in reverse order

```
USER(1): (reverse '(what a nice list))
(LIST NICE A WHAT)

USER(2): (reverse '((what a nice list)))
((WHAT A NICE LIST))
```

- `length` takes a list as argument, and returns an integer which is the number of elements in the argument list

```
USER(3): (length '(what a nice list))
4

USER(4): (length '((what a nice list)))
1
```

Numerical processing in LISP

- ✔ LISP has many built-in functions for manipulating numbers:

```
USER: (+ 1 2 3)
6
USER: (- 2 3)
-1
USER: (- 2 3 4)
-5
USER: (- 2)
-2
USER: (* 4 5 6)
120
USER: (/ 4 5)
4/5
USER: (/ 4 5 6)
2/15
USER: (/ 4)
1/4
USER: (float (/ 4 5))
0.8
USER: (round (/ 4 5))
1
-1/5
USER: (ceiling (/ 4 5))
1
-1/5
USER: (floor (/ 4 5))
0
4/5
USER: (mod 13 4)
1
USER: (rem 13 4)
1
```

Numerical processing in LISP

```
USER: (max 1 2 3 4 5)
5

USER: (min 1 2 3 4 5)
1

USER: (log 2)
0.6931472

USER: (exp 1)
2.7182817

USER: pi
3.141592653589793d0

USER: (cos pi)
-1.0d0

USER: (sin pi)
0.0d0

USER: (sqrt 2)
1.4142135

USER: (sqrt -1)
#c(0.0 1.0)

USER: (abs -1)
1

USER: (expt 2 10)
1024
```

Defining functions in LISP

- ✔ To define a function in LISP, use the `defun` construct, which has this form:

```
(defun function-name parameter-list
       expression1
       expression2
       ...
       expressionN )
```

  where

  function-name is a symbol

  parameter-list is a list of symbols

  expression1, ... , expressionN are LISP expressions which make up the "body" of the function

- ✔ `defun` is a macro, not a function: it does not evaluate its arguments
- ✔ `defun` returns as value the symbol *function-name*
- ✔ `defun` has as "side-effect" associating a function definition with the symbol *function-name*

Defining functions in LISP

- Example:

```
USER: (defun foo (x y bar)
          (crunch y x)
          (smash bar)
          (list (length (rest x)) y))
FOO
```

Now the symbol `foo` has a function definition associated with it; this is a function of 3 arguments:

```
USER: (foo arg1 arg2 arg3)
```

- The symbol `foo` is checked to see if it has a function definition
- The arguments `arg1, arg2, arg3` are evaluated
- The values are assigned ("bound") to the parameters `x, y, bar` respectively
- The expressions in the body of the function definition are evaluated in order in the context of these *lexical variable bindings*
- The value of the last expression in the body of the function definition of `foo` is returned as the value of the expression `(foo arg1 arg2 arg3)`

Defining functions in LISP

- ✔ Function definitions can be placed in a file and loaded into LISP using the `load` function

```
USER: (load "homework.cl")
```

- ✔ Example:
  - ✗ Define a function called `both-ends` which takes a list as argument and returns a two-element list made of the first and last elements of the argument list. Put the definition in a file named "bothends.cl".

Contents of the file bothends.cl:

```
(defun both-ends (list)
   (list (first list) (first (reverse list))))
```

  - ✗ Load the function definition into LISP

```
USER(1): (load "bothends")
; Loading /a/sdcc5/u/disk01/cs162f/cs162f/bothends.cl.
T
USER(2): (both-ends '(a b c d))
(A D)
USER(3): (both-ends '(a))


(A A)

USER(4): (both-ends nil)


(NIL NIL)
```

Commenting LISP functions

- ✔ The LISP `read` function ignores everything from a semicolon "`;`" to the end of the line
- ✔ Therefore, comments in LISP programs can begin with a semicolon
- ✔ LISP hackers usually use this convention:
  - ✗ comments outside of function definitions start with 3 semicolons
  - ✗ comments on their own line within function definitions start with 2 semicolons
  - ✗ comments at the end of a line of LISP start with 1 semicolon

```
;;; function to compute Einstein's famous formula
(defun einstein (m)
    (let ((c 300000000.0)) ;speed of light in m/sec
        ;; m is mass in kg
        ;; compute energy in joules
        (* m c c)))
```

Commenting LISP functions

- A function definition can also include a *documentation string*

- A documentation string is a string placed after the argument list in a function definition

```
(defun einstein (m)
    "(einstein m) returns the energy equivalent in
     joules of mass m in kilograms,
     according to e = mc^2"
    (let ((c 300000000.0)) ;speed of light in m/sec
       (* m c c)))
```

- The documentation string for a function can be inspected by using the `documentation` function in LISP

```
USER: (documentation 'einstein 'function)
"(einstein m) returns the energy equivalent in
     joules of mass m in kilograms,
     according to e = mc^2"
```

Predicates in LISP

- ✔ A predicate is a function that returns a value to be taken as "true" or "false"
- ✔ In LISP, the constant `nil` is taken to be equivalent to "false"
- ✔ In LISP, the constant `t` is taken to be equivalent to "true"
  - ✗ in most contexts, any value that is not `nil` will work

Predicates in LISP

✔ Equality predicates

```
USER: (equal (+ 2 2) 4)
T
USER: (equal (+ 2 2) 5)
NIL
USER: (equal '(this is a list) '(THIS IS A LIST))
T
USER: (equal '(a list) (reverse (reverse '(a list))))
T
USER: (equal 4 4.0)
NIL
USER: (= 4 4.0)
T
USER: (eq (+ 2 2) 4)
T
USER: (eq 'a 'a)
T
USER: (eq '(a) '(a))
NIL


; eq checks if its arguments are the same symbol

; = checks if its arguments are numerically the same

; equal checks if its arguments are the same expression
```

Predicates in LISP

- ✔ Data-type predicates
  - ✗ Since LISP is an implicitly-typed language, it is useful to be able to explicitly check the type of a variable value

```
; atom checks if its argument is an atom
   USER: (atom pi)
   T
   USER: (atom `(pi))
   NIL

; numberp checks if its argument is a number
   USER: (numberp pi)
   T
   USER: (numberp `pi)
   NIL

; symbolp checks if its argument is a symbol
   USER: (symbolp pi)
   NIL
   USER: (symbolp `pi)
   T

; listp checks if its argument is a list
   USER: (listp pi)
   NIL
   USER: (listp `(pi))
   T
```

Predicates in LISP

✔ Numerical predicates

```
; = checks if its arguments are all numerically equal
   USER: (= 3 3 (/ 12 4) 3 3.0)
   T

; /= checks if its arguments are all numerically different
   USER: (/= 3 4 5 6 3)
   NIL

; < checks if its arguments are in increasing order
   USER: (< 1 5.5 (expt 3 2))
   T

; > checks if its arguments are in in decreasing order
   USER: (> pi 4 2)
   NIL

; evenp checks if its argument is even
   USER: (evenp (expt 2 10))
   T

; oddp checks if its argument is odd
   USER: (oddp (* 5 7 11 3 4 7)
   NIL

; (zerop n) is equivalent to (= n 0)
; (minusp n) is equivalent to (< n 0)
; (plusp n) is equivalent to (> n 0)
```

Predicates in LISP

✔ Additional predicates

```
; null checks if its argument is the empty list

    USER: (null nil)
    T

    USER: (null 'a)
    NIL


; endp checks if its argument is a list and is empty

    USER: (endp nil)
    T

    USER: (endp 'a)
    ERROR: Illegal argument to endp: A


; member checks if its first argument is an element of its
; second argument

    USER: (member 'father '(my mother does not like me))
    NIL

    USER: (member 'mother '(my mother does not like me))
     (MOTHER DOES NOT LIKE ME)
```

Predicates in LISP

✔ Logically combining predicates

- ✗ `and` returns `nil` if any of its arguments are `nil`
- ✗ arguments are evaluated left to right
  - if one evaluates to `nil`, none of the other arguments are evaluated, and `nil` is returned
  - if none evaluate to `nil`, the value of the last argument is returned
  - can be used for execution control

```
(and (numberp x) (< x 10) (> x 3) (print x))
```

- ✗ `or` returns `nil` if all of its arguments are `nil`
- ✗ arguments are evaluated left to right
  - if one evaluates to something not `nil`, that value is returned, and none of the other arguments are evaluated
  - if all evaluate to `nil`, nil is returned
  - can be used to return interesting values

```
(or (memberp x L) (memberp y L) (memberp z L))
```

- ✗ `not` returns `t` if its argument is `nil`, and returns `nil` otherwise
  - equivalent to `null`