Common Lisp

What is Lisp?

- Designed in 1956 by John McCarthy
- It’s way more than ML with lots of parenthesis
- Became named for what is was good at:
  - LISt Processing
  - Lots of Irritating Superfluous Parenthesis

Two Black Boxes

Most programming languages have a single "black box" which has three components: lexical analyzer, parser, evaluator.

Lisp has two black boxes:

1.) Reader: translate text into LISP objects

2.) Evaluator: implements semantics of language

Two Black Boxes (cont.)

Allows you to…

- You can “print” code and “read” it back in
- Semantics of language is defined in terms of trees of Lisp objects, so generating its easy to generate code within the language
- Generate code by manipulating existing data

S-Expressions

- S-expressions are Lisp objects
- Composed of *lists* and *atoms*
- Valid identifiers consist of: A-Z 0-9 + - / & %

    and is NOT case sensitive
- Like most languages that support objects, Lisp passes them around by reference
- Some s-expressions are called "forms"
- Not all s-expressions are valid:

    (foo 1 2)

    ("foo" 1 2)

Atoms

Numbers:

17

1/7

1.7

-17

+17

1.7d0

1.7e-4

Atoms (cont.)

Strings:

“foo” => “foo”
“fo\o” => “foo”
“fo\\o” => “fo\o”
“fo\”o” => “fo\”o”

Chars:

#\a
#\+

Functions

Normally functions are defined using the DEFUN macro:

```
(defun name (parameter*)
    "optional documentation string"
    body-form*)
```

Definition:

```
(defun hello-world ()
    "useful documentation"
   (print  "hello world" ))
```

Invocation:

```
(hello-world) =>  "hello world"
(hello-world ()) => ERROR
```

Lists

Technically, list is a subtype of sequence, which is also the base type of other common lisp data structures

- ()
- (1 2 3)
- (“foo” “bar”)
- (x y z)
- (foo 1 2 3)

Lists (cont.)

P-lists where :key is a keyword

(this is NOT a hash table!)

```
(list :a 1 :b 2 :c 3) => (:A 1 :B 2 :C 3)
```

```
(getf (list :a 1 :b 2 :c 3) :b) => 2
(setf (getf(list :a 1 :b 2) :b) 3) => (:A 1 :B 3)
```

cons

Actually, there are no lists in lists

```
(cons 1 2) => (1 . 2)
(1, 2) => (1, 2)

(cons 1 (cons 1 2) => (1 . (2 . 3)
(1, (1, 2)) => (1, (2, 3))
```

cars and cdrs

```
(car(cons(1 (cons (2, 3))) => 1
hd [1, 2, 3] => 1
(car (list 1 2 3)) => 1

(cdr(cons(1 (cons (2, 3))) => (2 . 3)
tl [1, 2, 3] => [2, 3]
(cdr (list 1 2 3)) => (2 3)
```

But Lisp is better:

```
hd tl tl [1, 2, 3, 4] => 3
(caddr (list 1 2 3 4)) => 3
```

Variables

Local:

```
(setf x 10)
```

Globals:

```
(defparameter *db* ()  “documentation” )
(defvar *db* (1 2 3)  “documentation” )
```

Constants:

```
(defconstant +c+ 11  “documentation” )
```

Special Parameters

Optional parameters:

```
(defun foo (a b &optional c d)
   (list a b c d))
(defun foo (a b &optional (c 10))
   (list a b c)
```

Rest parameters :

```
(defun foo (&rest numbers)
   (+ numbers))
```

Keyword parameters:

```
(defun foo (&key a b c)
   (list a b c))

(foo :a 1 :b 2 :c 3) => (1 2 3)
```

Cool Stuff

You can return an s-expression instead of evaluating it! This is done by suppressing evaluation:

```
(1 2 3) => ERROR
`(1 2 3) => (1 2 3)
``(1 2 3) => `(1 2 3)
```

You can also "unsuppress" evaluation:

```
``,(1 2 3) => `(1 2 3)
,(1 2 3)=> ERROR
```

Some Useful Functions

```
(first  ‘(1 2 3 4 5)) => 1
(rest  ‘(1 2 3 4 5)) => (2 3 4 5)
(last  ‘(1 2 3 4 5)) => 5

(append  ‘(1 2 3)  ‘(4 5 6) ) => (1 2 3 4 5 6)
[1, 2, 3] @ [4, 5, 6] => [1, 2, 3, 4, 5, 6]

`(1 2 ,@(2 3 4)) => (1 2 3 4)
1::2::[3, 4] => [1, 2, 3, 4]
```

More Useful Functions

```
(map `list #`+  `(1 2 3) `(1 1 1)) => (2 3 4)
map (fn x=>x+1) [1, 2, 3] => [2, 3, 4]

(reduce #' +   #(1 2 3)) => 6
foldl op+ 0 [1, 2, 3] => 6

(random 10)
(quote (1 2 3))
(atom `(1 2 3))
(null ())
```

Sequence Functions

```
(length s)
(elt s 0)
(setf (elt s 0) 99)
(count 1 ‘(1 2 3 1)) => 2
(remove 1 ‘(1 2 3 1)) = (2 3)
(substitute 10 1  ‘(1 2 3 1)) => (10 2 3 10)
(find 1 ‘(1 2 3 1)) => T
(find 10 ‘(1 2 3 1)) => NIL
(position 1 ‘(1 2 3 1)) => 0
(concatenate ‘vector ‘(1 2 3) ‘(4 5 6)) => #(1 2 3 4 5 6)
(sort  ‘(4 1 2 3) #’ < ) => ( 1 2 3 4)
(merge  ‘vector #(1 3 5) #(2 4 6) #’ <) => #(1 2 3 4 5 6)
```

Sequence Predicates

```
(every #'evenp '(1 2 3 4 5)) => NIL

(some #'oddp '(1 2 3 4 5)) => T

(notany #'> '(1 2 3) '(1 2 3)) => T

(notevery #'> '(1 2 3) '(1 2 3)) => T
```

Math Stuffs

```
(+ 1 2) => 3
(* 1 2) => 2
(- 1 2) => -1
(/ 1 2) => 1/2
(= 1 1) = > T
(< 2 3) => T
(>= 2 3) => NIL
```

Math Stuffs (cont.)

```
(log x)
(exp x)
(sin x)
(floor x)
(max 1 2 3)
(min a b c)
```

Control Constructs

```
(if condition then-form [else-form])
(if t (print “Yup” ) (print “Nope” ))

(when condition &rest body)
(when t (print “Hello” ) (print “World” ))

(unless condition &rest body)
(unless t (print “Hello” ) (print “World” ))

(let (variable*) body-form*)
(let ((x 1) (y 2) z) (list x y z))

(not condition), (and condition*), (or condition *)
```

Loops

```
(dolist (x `(1 2 3)) (print x))
(dotimes (x 5) (print x))



(do (variable-definition*)
  (end-test-form result-form*)
  statement*)

(do ((i 0 (+ 1 i)))
  ((= i 5))
  (print i))
```

Loops (cont.)

```
(loop body-form*)

(loop for i from 1 to 5 collecting i) => (1 2 3 4 5)

(loop for i from 1 to 5 summing i) => 15

(loop
    (when (> (get-universal-time) *some-future-date*)
          (return))
    (print "waiting..." )
    (sleep 100))
```

Vectors

Vectors default to a fixed size, but you can provide keywords to change how its working

```
:initial-element   5
:element-type   NUMBER
:fill-pointer   0
:adjustable   t
```

Vectors (cont.)

(vector) => #()
(vector 1 2 3 => #(1 2 3)

(vector-pop v)
(vector-push ‘c v)

(make-array 5 :initial-element nil) => #(nil, nil, nil, nil, nil)
(make-array 5 :fill-pointer 0) => #()
(make-array 5 :fill-pointer 0 :adjustable t) => #()

All the sequence functions work for vectors too!

Hash Tables

```
(defparameter *h* (make-hash-table))

(gethash  ‘foo *h*) => NIL
(setf (gethash  ‘foo *h*)  ‘bar)
(gethash  ‘foo *h*) => BAR
```

I/O

Input:

```
(read-line)
(parse-integer (read-line))
```

Output:

```
(print “hello” )
(write-line “hello” )
(format t  “hello” )
(format t “hello ~a”  “world” )
```

Anonymous Functions

```
(lambda (parameters) body)

((lambda (x) x+1) 11) => 12
(fn x ==> x+1) 11 => 12
```

Macros

At first, LISP had no macros. And then it did.

```
(defmacro name (parameter*)
   "Optional documentation string."
  body-form*)
```

Macros accept and return ASTs. This means you can write macros with macros!

Macros (cont.)

```
(if condition then-form [else-form])
(if (< 2 3)  “Yup” )
(if (< 2 3)
  (progn
      ( “Yup” )
      ( “Yup, again” )))

(defmacro if2 (condition &rest body))
  `(if ,condition (progn ,@body)))

(if2 (< 2 3) ( “Yup” ) (  “Yup, again” ))
```

Classes

A simple class:

```
(defclass name (direct-superclass-name*)
    (slot-specifier*))

(defclass bank-account ()
    (customer-name
    balance))

(make-instance  ‘bank-account)
```

Another class:

```
(defclass bank-account()
    ((customer-name
    :initarg :customer-name)
    (balance
    :initarg :balance))

(make-instance  ‘bank-account
    :customer-name  “John Doe”
    :balance 1000)
```

Generics

Accessors:

```
(defun balance (account)
   (slot-value account  ‘balance))
```

To support subclasses you want:

```
(defgeneric balance (account))

(defmethod balance ((account bank-account))
   (slot-value account  ‘balance))
```

Mutators:

```
(defgeneric (setf balance ) (value account))

(defmethod (setf balance) (value (account bank-account))
   (setf (slot-value account  ‘balance) value))
```

Inheritance

All members of a class and functions that operate on that class are usable in subclasses.

```
(def class foo()
  ..)
(def class bar (foo)
  ..)
```

…can have multiple, direct superclasses!

CALL-NEXT-METHOD makes a call "up" to the next most specific function.

call-with-current-continuation

Although only supported by other dialects of lisp, the call-with-current-continuation function (or call/cc) provides users with a totally new way of doing things.

Scheme Example:

```
(define (f return)
    (return 2)
    3)

(display (f (lambda (x) x)))

(display (call-with-current-continuation f))
```

So Why Lisp?

- Emacs (which almost as good as [gk]edit)
- A LISP ninja can code faster than a C++ ninja
- C++ doesn’t even have hash tables in the STL
- It’s better than ML
- Macros make everything cleaner, shorter, and easier
- Writing code that writes code is better than writing code that doesn’t write code

Dialect of Lisp

- Common Lisp
- Emacs Lisp
- Scheme
- Arc
- Dylan

References

- Practical Common Lisp by Peter Seibel

- Wikipedia

http://en.wikipedia.org/wiki/Lisp_programming_language