Common Lisp II

Input and Output

• Print is the most primitive output function
  > (print (list 'foo 'bar))
  (FOO BAR)
  (FOO BAR)

• The most general output function in CL is format which takes two or more arguments:
  – the first indicates where the input is to be printed,
  – the second is a string template,
  – the remaining arguments are objects whose printed representations are to be inserted into the template:
  > (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
  2 plus 3 equals 5.
  NIL

Read

• The standard function for input is read.
• When given no arguments, it reads from the default place, which is usually standard input.
  > (defun ask (string)
      (format t "~A" string)
      (read))
  ask
  > (ask "How old are you? ")
  How old are you? 29
  29

Local Variables

• One of the most frequently used operators in CL is let.
• This allows local variables to be used in a function.
• A let expression has two parts.
  – First comes a list of instructions for creating variables, each of the form var or (var expression).
  Local variables are valid within the body of the let.
  – After the list of variables and values comes the body of expressions, which are evaluated in order.
  > (let ((x 100) (y 200))
    (print (+ x y))
    (setq x 200)
    (print (+ x y))
    'foo)
  300
  400
  foo
A let example

> (defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
      val
      (ask-number)))
) ASK-NUMBER

> (ask-number)
  Please enter a number.  number
  Please enter a number.  (this is a number)
  Please enter a number.  52
  52

Global variables

• Global variables are visible throughout the program.
• Global variables can be created by giving a symbol and a value to defparameter or defvar.

> (defparameter *foo* 1)
*FOO*

> *foo*
1

> (defvar *bar* (+ *foo* 1))
*BAR*

> *bar*
2

Note: (defparameter v e) creates a global variable named v and sets its value to be e.
(defvar v e) is just like defparameter if no global variable named v exists. Otherwise it does nothing.

Global constants

• You can define a global constant, by calling defconstant.

> (defconstant +limit+ 100)
+LIMIT+

> (setf +limit+ 99)
*** - SETQ: the value of the constant +LIMIT+ may not be altered

• The plus-something-plus is a lisp convention to identify symbols as constants. Just like star-something-star is a lisp convention to identify global variables.

When in doubt

• When in doubt about whether some symbol is a global variable or constant, use boundp.

> (boundp '*foo*)
T

> (boundp 'fishcake)
NIL
Assignment

- There are several assignment operators in Common Lisp: set, setq and setf.
- The most general assignment operator is `setf`.
- We can use it to assign both local and global variables:
  ```lisp
  > (setf *blob* 89)
  89
  > (let ((n 10))
      (setf n 2)
      n)
  2
  ```

Setf

- You can create global variables implicitly just by assigning them values.
  ```lisp
  > (setf x (list 'a 'b 'c))
  (A B C)
  ```
- However, it is better lisp style to use defparameter to declare global variables.
- You can give setf any even number of arguments:
  ```lisp
  (setf a 1 b 2 c 3)
  ```
  is the same as:
  ```lisp
  (setf a 1)
  (setf b 2)
  (setf c 3)
  ```
- You can do more than just assign values to variables with setf.
- The first argument to setf can be an expression as well as a variable name.
- In such cases, the value of the second argument is inserted in the place referred to by the first:
  ```lisp
  > (setf (car x) 'n)
  N
  > (N B C)
  ```
Functional programming

- **Functional programming** means writing programs that work by returning values, instead of by modifying things.
- It is the dominant programming paradigm in Lisp.
- Must built-in lisp functions are meant to be called for the values they return, not for side-effects.

Examples of functional programming

- The function `remove` takes an object and a list and returns a new list containing everything but that object:
  ```lisp
  > (setf lst '(b u t t e r))
  (B U T T E R)
  > (remove 'e lst)
  (B U T T R)
  ```
- Note: remove does not remove an item from the list! The original list is untouched after the call to remove:
  ```lisp
  > lst
  (B U T T E R)
  ```
- To actually remove an item from a list you would have to use `setf`:
  ```lisp
  > (setf lst (remove 'e lst))
  ```
- Functional programming means, essentially, avoiding `setf`, and other assignment macros.

How remove could be defined

Here’s how remove could be defined:

```lisp
(defun remove (x list)
  (cond ((null list) nil)
        ((equal x (car list))
         (remove x (cdr list)))
        (t (cons (car list) (remove x (cdr list)))))
```

Note that it “copies” the top-level of the list.

Iteration

- When we want to do something repeatedly, it is sometimes more natural to use iteration than recursion.
- This function uses `do` to print out the squares of the integers from `start` to `end`:
  ```lisp
  (defun show-squares (start end)
    (do ((i (i start (+ i i)))
         (> i end) 'done)
        (format t "~A ~A~%" i (* i i)))
  ```
**do**

- The *do* macro is CL's fundamental iteration operator.
- Like *let*, *do* can create variables, and the first argument is a list of variable specifications. Each element is of the form: `(var initial update)` where `variable` is a symbol, and `initial` and `update` are expressions.
- The second argument to *do* should be a list containing one or more expressions.
  - The first expression is used to test whether iteration should stop. In the case above, the test expression is `(> i end)`.
  - The remaining expression in this list will be evaluated in order when iteration stops, and the value of the last will be returned as the value of the *do*, *done* in this example.
- The remaining arguments to *do* comprise the body of the loop.

**Dolist**

- CL has a simpler iteration operator for handling lists, *dolist*.

```lisp
(defun len (lst)
  "I calculate the length of lst"
  (let ((l 0))
    (dolist (obj lst) (setf l (+ l 1)))
    l))
```

- Here *dolist* takes an argument of the form *(variable expression)*, followed by a body of expressions.
- The body will be evaluated with *variable* bound to successive elements of the list returned by expression.

**eval**

- You can call Lisp's evaluation process with the *eval* function.

```lisp
> (setf s1 (cadr '(one two three)))
(CADR '(ONE TWO THREE))
> (eval s1)
TWO
> (eval (list 'cdr(car '((quote (a . b)) c))))
B
```

**Functions as objects**

- In lisp, functions are regular objects, like symbols, or strings, or lists.
- If we give the name of a function to *function*, it will return the associated object.
- Like *quote*, *function* is a special operator, so we don't have to quote the argument:

```lisp
> (defun add1 (n) (+ n 1))
ADD1
> (function +)
#<SYSTEM-FUNCTION +>
> (function add1)
#<CLOSURE ADD1 (N) (DECLARE (SYSTEM::IN -DEFUN ADD1)) (BLOCK ADD1 (+ N 1))>
```
• Just as we can use ‘ as an abbreviation for quote, we can use #’ as an abbreviation for function:

\[
\begin{align*}
> & \texttt{#'+} \\
& \texttt{<SYSTEM-FUNCTION +>} \\
\end{align*}
\]

• This abbreviation is known as sharp-quote.

• Like any other kind of object, we can pass functions as arguments.

• One function that takes a function as an argument is apply.

---

### Apply

• `Apply` takes a function and a list of arguments for it, and returns the result of applying the function to the arguments:

\[
\begin{align*}
> & \texttt{(apply \texttt{#'+} \texttt{(1 2 3)})} \\
& 6 \\
\end{align*}
\]

• It can be given any number of arguments, so long as the last is a list:

\[
\begin{align*}
> & \texttt{(apply \texttt{#'+} 1 2 \texttt{(3 4 5)})} \\
& 15 \\
\end{align*}
\]

• A simple version of apply could be written as follows

\[
\begin{align*}
& \texttt{(defun apply (f list) (eval (cons f list)))} \\
\end{align*}
\]

---

### Funcall

• The function `funcall` is like `apply` but does not need the arguments to be packaged in a list:

\[
\begin{align*}
> & \texttt{(funcall \texttt{#'+} 1 2 3)} \\
& 6 \\
\end{align*}
\]

• It could be written as:

\[
\begin{align*}
& \texttt{(defun funcall (f &rest args) (eval (cons f args)))} \\
\end{align*}
\]

---

### Lambda

• The `defun` macro creates a function and gives it a name.

• However, functions don’t have to have names, and we don’t need `defun` to define them.

• We can refer to functions literally by using a `lambda expression`. 
Lambda expression

• A **lambda expression** is a list containing the symbol `lambda`, followed by a list of **parameters**, followed by a **body** of zero or more expressions:

  ```lisp
  > (setf f (lambda (x) (+ x 1)))
  #<CLOSURE :LAMBDA (X) (+ X 1)>
  > (funcall f 100)
  101
  ```

• A lambda expression can be considered as the name of a function.

• Like an ordinary function name, a lambda expression can be the first element of a function call:

  ```lisp
  > ((lambda (x) (+ x 100)) 1)
  101
  ```

• and by affixing a sharp-quote to a lambda expression, we get the corresponding function:

  ```lisp
  > (funcall #'(lambda (x) (+ x 100)) 1)
  101
  ```

Types

• In CL **values** have types, not **variables**.

• You don’t have to declare the types of variables, because any variable can hold objects of any type.

• Though type declaration is never required, you may want to make them for reasons of efficiency.

• The built-in CL types form a hierarchy of subtypes and supertypes.

• The type `t` is the supertype of all types, so everything is of type `t`.

```lisp
\[
\begin{array}{c}
t \\
\uparrow \\
\text{atom} \\
\uparrow \\
\text{number} \\
\uparrow \\
\text{real} \\
\uparrow \\
\text{rational} \\
\uparrow \\
\text{integer} \\
\uparrow \\
\text{fixnum} \\
\uparrow \\
27
\end{array}
\]
```

```lisp
> (typep 27 't) 
T
> (typep 27 'atom) 
T
> (typep 27 'number) 
T
> (typep 27 'real) 
T
> (typep 27 'rational) 
T
> (typep 27 'integer) 
T
> (typep 27 'fixnum) 
T
> (typep 27 'vector) 
NIL
```