# Introduction to Lisp: A Sample Run

Here is a sample session with sbcl. The main points are:

* Lisp is an interpreted language (which also comes with a compiler) where functions are defined by

(defun function-name parameter-list   
 body)

This definition tells Lisp a function under the given name is defined.

* Lisp has an environment; it knows, e.g. builtin functions, as well as user-defined functions.
* Numbers are known to Lisp, they are considered constant functions (but real numbers are approximated).
* When evaluating an expression

(e1 e2 ... en)

Lisp always interprets it as a function application where e1 is a function and e2 ... en are its arguments. If this is not what you intended, use the function quote, as in

(quote (e1 e2 ... en))

or simply

'(e1 e2 ... en)

The function quote simply returns its argument:

'(e1 e2 ... en) => (e1 e2 ... en)

* Empty list is represented by () and nil, both of which are considered the same atom in Lisp. In addition, nil represents the boolean value FALSE. Everything else is considered representing TRUE.
* Ignore the "dotted notation" for now.

The following is a sample run of Lisp using sbcl.

The things I typed are on yellow background.

Comments that I added later have green background.

% sbclThis is SBCL 1.1.14.debian, an implementation of ANSI Common Lisp.  
More information about SBCL is available at <http://www.sbcl.org/>.  
SBCL is free software, provided as is, with absolutely no warranty.  
It is mostly in the public domain; some portions are provided under  
BSD-style licenses.See the CREDITS and COPYING files in the  
distribution for more information.  
\* hello \* is the prompt sign, I typed hello but...  
debugger invoked on a UNBOUND-VARIABLE in thread#<THREAD "main thread" RUNNING {1002A8AF73}>:  
  The variable HELLO is unbound. Lisp doesn't know what hello is and throws me into the debugger. Also notice: Lisp turns code into uppercase  
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.  
restarts (invokable by number or by possibly-abbreviated name):  
0: [ABORT] Exit debugger, returning to top level.  
(SB-INT:SIMPLE-EVAL-IN-LEXENV HELLO #<NULL-LEXENV>)  
0] help I followed the advice to "Type HELP for debugger help"  
The debug prompt is square brackets, with number(s) indicating the current  
  control stack level and, if you've entered the debugger recursively, how  
  deeply recursed you are.  
Any command -- including the name of a restart -- may be uniquely abbreviated.  
The debugger rebinds various special variables for controlling i/o, sometimes  
  to defaults (much like WITH-STANDARD-IO-SYNTAX does) and sometimes to  
  its own special values, based on SB-EXT:\*DEBUG-PRINT-VARIABLE-ALIST\*.  
Debug commands do not affect \*, //, and similar variables, but evaluation in  
  the debug loop does affect these variables.  
SB-DEBUG:\*FLUSH-DEBUG-ERRORS\* controls whether errors at the debug prompt  
  drop you deeper into the debugger. The default NIL allows recursive entry  
  to debugger.  
Getting in and out of the debugger:  
  TOPLEVEL, TOP  exits debugger and returns to top level REPL  
  RESTART        invokes restart numbered as shown (prompt if not given).  
  ERROR          prints the error condition and restart cases.  
  The number of any restart, or its name, or a unique abbreviation for its  
   name, is a valid command, and is the same as using RESTART to invoke  
   that restart.  
Changing frames:  
  UP     up frame         DOWN     down frame  
  BOTTOM bottom frame     FRAME n  frame n (n=0 for top frame)  
Inspecting frames:  
  BACKTRACE [n]  shows n frames going down the stack.  
  LIST-LOCALS, L lists locals in current frame.  
  PRINT, P       displays function call for current frame.  
  SOURCE [n]     displays frame's source form with n levels of enclosing forms.  
Stepping:  
  START Selects the CONTINUE restart if one exists and starts  
        single-stepping. Single stepping affects only code compiled with  
        under high DEBUG optimization quality. See User Manual for details.  
  STEP  Steps into the current form.  
  NEXT  Steps over the current form.  
  OUT   Stops stepping temporarily, but resumes it when the topmost frame that  
        was stepped into returns.  
  STOP  Stops single-stepping.  
Function and macro commands:  
 (SB-DEBUG:ARG n)  
    Return the n'th argument in the current frame.  
 (SB-DEBUG:VAR string-or-symbol [id])  
    Returns the value of the specified variable in the current frame.  
Other commands:  
  RETURN expr  
    Return the values resulting from evaluation of expr from the  
    current frame, if this frame was compiled with a sufficiently high  
    DEBUG optimization quality.  
  RESTART-FRAME  
    Restart execution of the current frame, if this frame is for a  
    global function which was compiled with a sufficiently high  
    DEBUG optimization quality.  
  SLURP  
    Discard all pending input on \*STANDARD-INPUT\*. (This can be  
    useful when the debugger was invoked to handle an error in  
    deeply nested input syntax, and now the reader is confused.)  
(The HELP string is stored in \*DEBUG-HELP-STRING\*.)  
0] top I read the part about how to exit the debugger. Now let's try some valid Lisp.  
\*  'hello  
HELLO That's better. Quoting an expression tells Lisp to NOT evaluate it  
\* (quote hello) Same thing, using quote in the normal way of calling a function in Lisp  
HELLO  
\* (+3 4) I make another mistake. It reads the + and the 3 as a single thing, the number +3. Then complains that 3 is not a valid function name  
; in: 3 4  
;     (3 4)  
;   
; caught ERROR:  
;   illegal function call  
;   
; compilation unit finished  
;   caught 1 ERROR condition  
debugger invoked on a SB-INT:COMPILED-PROGRAM-ERROR in thread  
#<THREAD "main thread" RUNNING {1002A8AF73}>:  
  Execution of a form compiled with errors.  
Form:  
  (3 4)  
Compile-time error:  
  illegal function call  
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.restarts (invokable by number or by possibly-abbreviated name):  
  0: [ABORT] Exit debugger, returning to top level.  
((LAMBDA ()))  
0] top get back out o the debugger again  
\* (+ 3 4)  
7 that's better.  
\* 44 integers are constant functions in Lisp  
\* 5.675.67 floating point numbers, too  
\* (- 5.678 2.444)subtraction3.234  
\* (first '(1 2 3))1  
\* (car '(1 2 3))1  
\* (car '(a b c))A again, converted into uppercase. Lisp is not case-sensitive  
\* (car '((a b) c (d)))(A B)  
\* (car (a b c))mistake. Without quote, it thinks a is a function name and b and c are the function arguments; in: CAR (A B C)  
; (A B C)  
;   
; caught STYLE-WARNING:  
; undefined function: A  
;   
; caught WARNING:  
; undefined variable: B  
;   
; caught WARNING:  
; undefined variable: C  
;   
; compilation unit finished  
; Undefined function:  
; A  
; Undefined variables:  
; B C  
; caught 2 WARNING conditions  
; caught 1 STYLE-WARNING conditiondebugger invoked on a UNBOUND-VARIABLE in thread  
#<THREAD "main thread" RUNNING {1002A8AF73}>:  
 The variable B is unbound.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.restarts (invokable by number or by possibly-abbreviated name):  
 0: [ABORT] Exit debugger, returning to top level.((LAMBDA ()))  
0] top\* (cdr '(1 2 3))(2 3)  
\* (cdr '())Caution: In Common Lisp it is allowed to call car and cdr on empty lists. The result is nil

NIL \* (eq nil '()) nil and () are considered the same atom  
T  
\* (eq nil ()) quote in front of () is not necessary  
T  
\* (atom 'sillyName154)  
T  
\* (atom 78)  
T  
\* (atom '(1 2))  
NIL  
\* (atom ())  
T  
\* (null nil)  
T  
\* (null ())  
T  
\* (null '(1))  
NIL  
\* (eq 'a 'a)  
T  
\* (eq '(1) '(1)) eq only returns T if both sides are the same atom  
NIL  
\* (equal '(1) '(1)) equal compares nested expressions  
T  
\* (defun append (L1 L2) trying to define append, but it is already a built-in function  
     (if (null L1)  
         L2  
        (cons (car L1) (append (cdr L1) L2))  
     )  
 )

STYLE-WARNING: redefining COMMON-LISP:APPEND in DEFUN

STYLE-WARNING: redefining COMMON-LISP:APPEND in DEFUN  
debugger invoked on a SYMBOL-PACKAGE-LOCKED-ERROR in thread  
#<THREAD "main thread" RUNNING {1002A8AF73}>:  
  Lock on package COMMON-LISP violated when setting fdefinition of APPEND while  
  in package COMMON-LISP-USER.  
See also:  
  The SBCL Manual, Node "Package Locks"  
  The ANSI Standard, Section 11.1.2.1.2  
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.  
restarts (invokable by number or by possibly-abbreviated name):  
  0: [CONTINUE      ] Ignore the package lock.  
  1: [IGNORE-ALL    ] Ignore all package locks in the context of this operation.  
  2: [UNLOCK-PACKAGE] Unlock the package.  
  3: [ABORT         ] Exit debugger, returning to top level.  
(PACKAGE-LOCK-VIOLATION #<PACKAGE "COMMON-LISP"> :SYMBOL APPEND :FORMAT-CONTROL "setting fdefinition of ~A" :FORMAT-ARGUMENTS (APPEND))  
0] top  
\* (append '(1 2) '(a b))  
(1 2 A B)  
\* (append '(1 2) (append '(a b) '(w v)))  
(1 2 A B W V)

\* (defun fact (N)

    (if (or (eq N 0) (eq N 1))   
        1   
        (\* N (fact (- N 1)))  
    )  
)  
FACT

\* (fact 4)

24

\* (fact -1) Our program causes an infinite recursion when called with negative numbers

INFO: Control stack guard page unprotected  
Control stack guard page temporarily disabled: proceed with caution  
debugger invoked on a SB-KERNEL::CONTROL-STACK-EXHAUSTED in thread  
#<THREAD "main thread" RUNNING {1002A8AF73}>:  
  Control stack exhausted (no more space for function call frames).  
This is probably due to heavily nested or infinitely recursive function  
calls, or a tail call that SBCL cannot or has not optimized away.  
PROCEED WITH CAUTION.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.  
restarts (invokable by number or by possibly-abbreviated name):  
  0: [ABORT] Exit debugger, returning to top level.

(SB-KERNEL::CONTROL-STACK-EXHAUSTED-ERROR)

(SB-KERNEL::CONTROL-STACK-EXHAUSTED-ERROR)  
0] top  
INFO: Control stack guard page reprotected  
\* (defun xcount (L)  
    (if (null L)  
        0  
        (+ 1 (xcount (cdr L)))  
    )  
 )  
XCOUNT

\* (xcount ' (1 2 3 4))

4

\* (trace xcount) Use trace to see your function calls and results

(XCOUNT)

\* (xcount '(1 2 3 4))

  0: (XCOUNT (1 2 3 4))  
    1: (XCOUNT (2 3 4))  
      2: (XCOUNT (3 4))  
        3: (XCOUNT (4))  
          4: (XCOUNT NIL)  
          4: XCOUNT returned 0  
        3: XCOUNT returned 1  
      2: XCOUNT returned 2  
    1: XCOUNT returned 3  
  0: XCOUNT returned 4  
4