Computing Science
Kyoto Common Lisp

(by Judith A. Gartaganis)
(re-formatted by Oliver Weishaar)

References:

Common Lisp Craft	by R. Wilensky
Kyoto Common LISP *Report*	by T. Yuasa and M Hagiya

1 Introduction

1.1 Getting Started

To run the LISP interpreter, type:

kcl

KCL will print a welcome message and prompt for input with the character >. At the present time, KCL is available on **manning** and on the Suns.

1.2 Input

The standard input (default) is the keyboard. In this input mode, you may enter your entire program and execute it. If your program is contained in a file, the input can be redirected using the following command format:

kcl < myin

Remember, *myin* must contain all the commands necessary to run your program including the *exit* command.
If you wish to read in some source code and then continue to use the keyboard to enter the remaining code, load the file containing parts of the source code using the following command sequence (the quotation marks around the file name are essential):

kcl
(load "myin")

1.3 Output

The standard output (default) is the terminal screen. In this output mode, results from the KCL program will be displayed on the screen. If you wish to collect output in a file, output can be redirected using the following command format:

kcl > myout

All output will appear in the file *myout*, and nothing will appear on the screen.
Redirection of both input and output is allowed in a single command:

kcl < myin > myout

1.4 Filenames

KCL translates all characters to upper case. If a filename is in upper case, however, it must be specifically entered in capital letters for KCL. When the *(load "myin")* command is invoked, KCL looks for files of specific types in a particular order:

First, KCL looks for myin.o
If unsuccessful, KCL looks for myin.lsp
If still unsuccessful, KCL looks for myin
If still unsuccessful, KCL generates an error

1.5 Exiting KCL

To exit KCL, type the command:

$$> bye \text{ or } > CTRL - D$$

For emergency exit (when KCL seems to be stuck in an infinite loop, for example), type the following command sequence:

> CTRL-Z (emergency suspension of KCL*)*
kill %1 (kills KCL *process)*

2 Debugging and Program Checking

2.1 Getting Help

KCL provides on-line help which can be obtained by typing:

> (help)

If a command is entered improperly or if a function call terminates with an execution error, KCL automatically enters *DEBUG* mode. There are multiple debug levels indicated by the prompt, as shown by the following example.

>>	is the prompt for debug level 1
>>>>>	is the prompt for debug level 4

To exit a debug level, type *:q*. For example,

>>>: *q*	If in debug level 2, typing *:q* returns to
>>: *q*	level 1; another *:q* returns to the main
>	execution level

There are a number of special commands which can be executed in *DEBUG* mode to assist the user in determining what the problem is. A list of some of these commands will be displayed if the user types *:h* while in *DEBUG* mode.

2.2 Trace

Parts of the program execution can be monitored by *tracing* specific functions. The command format is as follows:

> *(trace myfun)*

When *myfun* is invoked, the tracer prints a list with the function name and arguments. On return from a traced function, the function name and returned values are printed. For example,

> (trace myfun)	
> (myfun arg1 ... argN)	
1 > (myfun arg1 ... argN)	LISP prints arguments on function entry.
< 1 (myfun value1 ... valueM)	LISP prints return values on function exit.

If *myfun* is recursive, there may be more than one printed trace. The number at the beginning of the output line indicates the nesting level. For example,

> (trace myfun)	
> (myfun arg1 ... argN)	
1 > (myfun arg1 ... argN)	
2 > (myfun arg1 ... argN)	LISP prints the new arguments for the recursive call.
< 2(myfun value1 ... valueM)	
< 1 (myfun value1 ... valueM)	

To obtain a list of all functions currently being traced, type

> *(trace)*

To cancel the trace on a particular function, type

> *(untrace myfun)*

To cancel all traces, type

> *(untrace)*

2.3 Step

This command allows the user to examine particular steps in an evaluation using step commands. To enter *step* mode, type

> *(step form)*

Remember, *form* is just a generic term for anything that is to be evaluated (usually a function call). In *step* mode, the evaluation is executed one step at a time with a pause in between in case the user wishes some information. To get a list of *step* commands, type *?* while in *step* mode. A brief outline of some of the *step* commands follows:

n or carriage return	execute the next step
s	finish the evaluation in the usual way, then return to single step mode
p	print current stuff
q	quit step mode
f myfun	execute in ordinary mode until "myfun" is called, then enter single-step mode
e form	evaluate in ordinary mode and print the value

3 Constructing and Executing LISP Programs

3.1 Program Construction

A LISP "program" consists of two modules:

The function definition part, which consists of user-defined functions.

The program command part, which consists of the functions calls necessary to carry out the assigned task.

The LISP interpreter works in interactive mode and all record of a particular session is lost once the interpreter is exited. Therefore, it is common practice to enter function definitions, and the like, in a file which can be loaded once the interpreter is invoked. The *load* command is very similar to the *#include* command in the C programming language.

The user can continue the session in interactive mode, issuing the function calls which constitute the program.

There are two ways you can avoid typing the program commands (that is, the function calls) every session.

1. In the function definition file, include a *main* function definition which includes all the program commands. Once the function definition is loaded, just invoke the main function with the usual function call.

2. Create a program file which includes as the first command the *(load myin)* instruction. Invoke KCL with the input redirected to this program file. Remember to include the *(bye)* command at the end of the file.

3.2 Saving Output

To save the output from a LISP session, redirect the output to a file. You may define your own functions for printing output in particular formats.

3.3 Comments

Comments begin with a semicolon (;) and can extend over one line only. The LISP interpreter does not scan any characters in the rest of the line after a (;) is encountered. If comments extend over multiple lines, each line must begin with a (;). An example follows.

```
; This is an acceptable comment

> (car '(a b c))          ; So is this

; For comments extending over several lines,
; be sure to begin each line with the
; comment delimiter (;)
```

3.4 Compiling LISP Code

KCL allows users to compile LISP code into object modules that can be loaded with the *load rm command. The* LISP code is translated into C and then into object code. The command to do this is:

(compile-file "myin.lsp")

where "myin.lsp" represents the name of a LISP file to be compiled. The resulting object code will be placed in a **.o** file. In this example the result will be put into "myin.o."

The *.o* file can be loaded in the same manner as a **.lsp** file.

4 LISP Objects

The basic unit in LISP is an atom. The atom is indivisible and has one of the following forms:

- alphabetic string (up to 255 characters)
  - This type of atom can serve as a variable or function identifier.
  - Remember that the LISP interpreter converts everything to uppercase so variable names and functions should be consistently in lowercase.
  - LISP recognizes a blank as a delimiter rather than an alphabetic character. If you wish to include one or more blanks in a character string, enclose the whole string in double quotes ("...") or vertical bars (—...—). When LISP returns an alphabetic value, it does not remove the string delimiters so a user-defined function to print alphabetic strings which contain blanks may be desirable.
- numeric string
  - This type of atom serves as a value.
  - Both integer and floating point values can be specified.

The primary data structure in LISP is the s-expression. The s-expression has one of the following forms:

- atom
- (s-expression) - that is, a list
- (s-expression s-expression ...s-expression)

The following are all s-expressions:

```
3
x
(3)
(3 x)
((3) x)
((a b) (c d))
```

One other data structure must be mentioned, although it is not recommended for general use. It is the *cons* cell and has the form:

- s-expression.s-expression

The *cons* cell is formed, just as the name implies, by applying the *cons* function to a pair of s-expressions. Not all *cons* cells are lists, however, because true lists always have a right *nil* pointer at the last cell while a *cons* cell can have the form (a.b). *Cons* cells of this sort are formed when the second argument to the *cons* function is not a list. Be especially careful about indiscriminantly mixing use of the *cons* cell and the *list* in program command sequences.

5 Evaluation

The LISP interpreter performs computations by evaluating s-expressions. If the s-expression is an atom, LISP evaluates that atom and returns its value. If the s-expression is a list, LISP interprets the list as a function call. The function is evaluated and its value returned.

5.1 Suppressing Evaluation

To suppress evaluation, use the *quote* function. Several examples follow.

Expression	Value	Comments
> (quote s)	s	This function call warns the interpreter to suppress evaluation and take the "literal" value of what follows as the value of the function call.
> 's	s	Shorthand notation.
> (setq x 3)		Shorthand for (set (quote x) 3). The string x is set to the value 3.
> x	3	LISP evaluates x.
> 'x	x	The ' suppresses evaluation.
> (setq a 1)		
> (setq b 2)		
> (setq c 3)		
> (setq x '(a b c))		
> x	(a b c)	The ' in the setq command warned the interpreter to set the string x to the literal value (a b c).
> (setq x (a b c))		
> x	(1 2 3)	In the absence of the ', LISP evaluates (a b c) before setting the string x to that value.

5.2 Special Values

LISP provides several predefined values.

- NIL – false or empty
- T – true or non-empty

5.3 Executing Functions in LISP

A function call in LISP is simply of the following form:

(function_name parameter_1 parameter_2 ... parameter_N)

User-defined functions can be specified using the *defun* built-in function. User-defined functions are like subroutines in other languages, and should be collected together in the function definition file. The basic format of the *defun* command is shown below.

(defun f_name parameter_list statement_1 statement_2 statement_N)

Just as suggested, the parameter list must be a **list** and each statement is an s-expression.

The *lambda* built-in function can be used to create a nameless function which will be used only once. A *lambda* expression has the form shown below.

(lambda parameter_list statement_1 statement_2 ... statement_N)

The entire expression can be used anywhere a function name would be used. The use of a lambda expression is a way of providing a function definition directly, rather than have the interpreter look for its definition.

6 LISP Built-In Functions

The most commonly used LISP build-in functions are described briefly below. The reader is referred to the references for extra details.

Numeric Functions		
Syntax	**Parameters**	**Action**
(+ P1 P2 ... PN)	Each of P1 to PN must evaluate to a numeric value.	Returns P1 + P2 + ... + PN
(- P1)	P1 must evaluate to a numeric value	Returns -P1
(- P1 P2)	Both P1 and P2 must evaluate to numeric values	Returns P1 - P2
(* P1 P2 ... PN)	Each of P1 to PN must evaluate to a numeric value	Returns P1 * P2 * ... * PN
(/ P1 P2)	Both P1 and P2 must evaluate to numeric values	Returns P1 / P2 (floating point division if either P1 or P2 are floating point numbers, integer division with truncation otherwise)

Boolean-Type Functions		
Syntax	**Parameters**	**Action**
(zerop P1)	P1 must evaluate to a numeric value	Returns T if P1 = 0
(atom P1)		Returns T if P1 evaluates to an atom, NIL otherwise
(not P1)		Returns T if P1 evaluates to NIL, NIL otherwise
(null P1)		Returns T if P1 evaluates to NIL, NIL otherwise
(equal P1 P2)		Returns T if P1 and P2 evaluate to the same value, NIL otherwise
(eq P1 P2)		Returns T if P1 point to the same place or if they are two equal integers, NIL otherwise
(numberp (P1)	P1 must be an atom	Returns T if P1 evaluates to a numerical value, NIL otherwise

Input/Output Functions		
Syntax	**Parameters**	**Action**
(read)		Reads a single s-expression from the standard input, evaluates it and returns its value.
(princ P1)	P1 is evaluated	Evaluates P1 and sends its value to the output buffer. Note that this is just for storage. Returns the value of P1.
(terpri)		Flushes the output buffer by generating an end-of-line character. The contents of the output buffer are printed to standard output.
(print P1)	P1 is evaluated	Does a princ followed by a terpri. Returns the value of P1.

List-Related Functions		
Syntax	**Parameters**	**Action**
(length P1)	P1 must evaluate to a list	Returns the number of elements in P1. Remember, a list within a list counts as a single element.
(member P1 P2 ...PN)	P2 must evaluate to a list	Returns P2 if P1 of P2, NIL otherwise. The "equal" condition function is used. Remember, a list within a list is treated as a single element.
(list P1 P2 ...PN)		Returns a list with elements P1 to PN. P1 to PN are evaluated.
(car P1)	P1 must evaluate to a list	Returns the first element of P1.
(cdr P1)	P1 should evaluate to a list	Returns a **list** constructed by removing the first element from P1. Returns () if P1 does not evaluate to a list or if P1 has only one element. Remember, a list within a list counts as a single element.
(caaddr P1)		Nested car and cdr function calls are allowed. In this example, the shorthand notation represents (car(car(cdr(cdr P1)))).
(cons P1 P2)	If P2 evaluates to a list, returns a true list. If P2 does **not** evaluate to a list and does **not** evaluate to NIL, returns a cons cell which is not a true list (it will have the form P1.P2 where P2 does not have a NIL right pointer)	Returns a **list** with P1 as its car and P2 as its cdr. Note that the assigned value of P2 is not changed.
(cons P1 (list P2))	Returns a true list when P2 is an atom. Remember, (list P2) evaluates to a list	Use this command format to avoid forming a cons cell when P1 and P2 evaluate to atoms.

Control-Related Functions		
Syntax	**Parameters**	**Action**
(apply P1 P2)	P1 is a function name	The function P1 is executed on P2. P2 must have the form of the parameter list specified for P1. The value of the function call is returned.
(mapcar P1 P2 ... PN)	P1 is a function name. Each of P2 to PN must evaluate to a list.	The function P1 is applied in turn to elements of P2 to PN. The first call to P1 has parameters ((car P2) (car P3) ... (car PN)). The next call has parameters ((cadr P2) (cadr PN) ... (cadr PN)) and so on. Intuitively, mapcar is like an array processing function. For example, (mapcar '+ '(1 2 3) '(4 5 6) '(7 8 9)) would first perform (+ 1 4 7), then (+2 5 8), then (+ 3 6 9). The value returned by mapcar would be (12 15 18).
(and P1 P2 ... PN)		Logical AND: returns PN if none of P1 to PN evaluate to NIL, returns NIL otherwise.
(or P1 P2 ... PN)		Logical OR: returns the first of P1 to PN which evaluates to non-NIL, returns NIL is all of P1 to PN evaluate to NIL.
(cond (P1 S11 S12 ... S1N) (P2 S21 S22 ... S2N) ... (PN SN1 SN2 ... SNN))	P1 to PN are s-expressions, Sij is a statement	cond finds the first s-expression that is true (evaluating each one) and then executes the sequence of statements beside that s-expression. The cond function is analogous to an "if ... then, else if ... then, else ..." construct. The else default (to be performed when everything else fails) will have the form (t "else stuff here")

User-Defined Subroutines		
Syntax	**Parameters**	**Action**
(defun N P S1 S2 ... SN)	N is the function name. P is the parameter LIST. S1 to SN are the s-expressions which are the statements to be executed.	The effect is to associate with the name N the parameter list, P and statement sequence, S1 to SN.
(lambda P S1 S2 ... SN)	P is the parameter LIST. S1 to SN are the s-expressions which are the statements to be executed.	A nameless function is defined. P and S1 to SN are the associated parameter list and statement sequence.