Fall 2008

A Short Reference Manual for LISP

CMSC 420
Hanan Samet

This guide details a few important functions of FRANZ LISP and COMMON LISP.

Note, that all predefined (i.e., built-in) function and symbol names in FRANZ LISP are in lower case. That includes `nil` and `t`. In COMMON LISP, however, they are defined in upper case. The expression reader converts all symbols to upper case, so that you may use either upper or lower case in your programs.

Intrinsic Functions

These are the functions which come predefined, and are available at sign-on to LISP. This is not a complete list. Functions that are specific to FRANZ LISP are marked with [F], whereas functions that are specific to COMMON LISP are marked with [C].

1 S-expression manipulation

`(car arg)`	Returns the first element in list `arg`, or the left pointer of an s-expression (cons node).
`(cdr arg)`	Return the rest of list `arg`, or the right pointer of an s-expression (cons node).
`(cons a1 a2)`	Allocate and return a cons node that points to `a1` and `a2`.
`(list a1 a2 ...an )`	Return a list of all the arguments.
`(append a1 a2)`	Return the merge of the two lists `a1` and `a2`.
`(reverse arg)`	Reverses the list `arg`.
`(subst new old sexp)`	Return a copy of `sexp` with all `old`'s converted to `new`'s.

2 S-expression predicates

`(null arg)`	Return `t` if `arg` is `nil`; `nil` otherwise.
`(atom arg)`	Return `nil` if `arg` is a cons node; `t` otherwise.
`(eq a1 a2)`	Return `t` if `a1` and `a2` are the same pointer; `nil` otherwise.
`(equal a1 a2)`	Return `t` if `a1` and `a2` have the same structure; `nil` otherwise.
`(member a list)`	Return the sublist of `list` beginning with the first occurence of `a`; `nil` if not in `list`.

3 Logical functions

`(or a1 a2 ...an)`	Return `t` if any argument is non-`nil`; `nil` otherwise.
`(and a1 a2 ...an)`	Return `t` if all arguments are non-`nil`; `nil` otherwise.
`(not arg)`	If `arg` is `nil`, returns `t`; else returns `nil`.

4 Control and program functions

`(quote a)` or `'a`	Return `a` unevaluated.
`(cond (exp1 a1 a2 ...an) (exp2 b1 b2 ...bm) ... (expn z1 z2 ...zk))`	Evaluates all `exp` until one is non-`nil`, then evaluates all expressions that follow and returns the result of the last.
`(prog (v1 v2 ...vn) label1 (s-expr1) label2 (s-expr2) ... labelm (s-exprm))`	Acts like a sequential program with local variables `v1` to `vn`. Labels are optional for each line. If no return is encountered and control "drops off the end," then `nil` is returned.
`(go arg)`	In a `prog`, branch control to `s-expr`*i* preceded by label `arg`.
`(return arg)`	Only valid in a `prog`. Terminate `prog` and return the value of `arg`.

5 I/O Functions

`(read)`	Return as a value the next s-expression typed as input to the terminal.
`(readc)` [F]	Return next character as an atom.
`(read-char)` [C]	
`(print arg)`	Print `arg` to output.
`(patom arg)` [F]	Print `arg` to output. Differs from `print` in that the output is a bit more readable, e.g., strings are not enclosed in quotes.
`(terpr)` [F]	Print a newline character.
`(terpri)` [C]	
`(pp arg)` [F]	Print the definition of the symbol `arg`. If `arg` is a function, then its binding is pretty printed. `pp` can take multiple arguments.
`(symbol-function atom)` [C]	Print the function definition of the function bound to `atom`.
`(cprintf format args)` [F]	Analogous to the C language version.
`(load filename)`	Read in the file `filename` and evaluates all expressions in it, including function definitions. Values of expressions are not displayed.
`(include filename)` [F]	Same as load, except that the argument is not evaluated, so it shouldn't be quoted.

6 Arithmetic Functions

`(numberp n)`	Return `t` if `n` is a number; `nil` otherwise.
`(zerop n)`	Return `t` it `n` is zero; `nil` otherwise.
`(> n1 n2)` or `(greaterp n1 n2)` [F]	Return `t` if `n1` is greater than `n2`.
`(< n1 n2)` or `(lessp n1 n2)` [F]	Return `t` if `n1` is less than `n2`.
`(+ n1 n2 ...nn)` or `(plus n1 n2 ...nn)` [F]	Return the sum of all arguments.
`(* n1 n2 ...nn)` or `(times n1 n2 ...nn)` [F]	Return the product of all arguments.
`(- n1 n2)` or `(difference n1 n2)` [F]	Return the quantity `n1` minus `n2`.
`(/ n1 n2)` or `(quotient n1 n2)` [F]	Return `n1` divided by `n2`.
`(1+ x)` or `(add1 x)` [F]	Return `x + 1`.
`(1- x)` or `(sub1 x)` [F]	Return `x - 1`.

7 Function Definition and Value Assignment

`(setq x y)`	Set `x` to the value of `y`. `setq` allows any number of `x-y` pairs.
`(set x y)`	Like `setq`, except that `x` is evaluated to get an atom.
`(lambda arglist body)`	Return a nameless function with argument list `arglist` and body `body`. The body should be a list of expression, with the value of the last one being the value of the function.
`(defun name arglist body)`	Define a function with name `name`. Equivalent to `(setq name (lambda arglist body))`.
`(defun name fexpr (arg) body)` [F] `(defun.fexpr name (arg) body)` [C]	Define a function where upon invocation, the argument list is not evaluated, but passed instead as the binding of the only parameter `arg` (not a built-in function in COMMON LISP).

8 Atom Manipulation

`(gensym arg)`	Create atom with name `argnnnnn`, where `arg` is an atom and `nnnnn` is the number of times `gensym` has been called. `(gensym 'x)` returns `x00000`.
`(putprop atom value label)` [F] `(setf (get atom label) value)` [C]	Set the value of the property `label` of `atom`'s property list to `value`.
`(get atom label)`	Return the value of property `label` on the property list of `atom`.
`(remprop atom label)`	Remove the property `label` from the property list of `atom`.
`(plist atom)` [F] `(symbol-plist atom)` [C]	Return the property list of `atom` in the form `(p1 v1 p2 v2 ... pn vn)`, where `pi` is a property with value `vi`.
`(setplist atom list)` [F] `(setf (symbol-plist atom) list)` [C]	Set the property list of `atom` to `list`, which must be a list of the form `(p1 v1 p2 v2 ... pn vn)`.

9 Other useful functions

`(implode list)` [F]	Return the atom created by concatenating the first character of all the atoms in `list`.
`(explode atom)` [F]	Return the list of the `atom`'s characters (reverse of `implode`).
`(concat-symbols symbol1 symbol2)` [C]	Return a symbol with a name that is a concatenation of the names of `symbol1` and `symbol2` (not a built-in function).
`(nth number list)`	Return the element of `list` with index `number`, assuming zero-based indexing.
`(nthcdr number list)`	Return the result of applying `cdr` to the `list` `number` times.
`(length list)`	Return the number of elements in the top level of `list`.
`(trace fname ...)`	Turn on function tracing for function `fname`. Trace can take multiple arguments. A list of all functions being traced is returned.
`(untrace fname ...)`	Turn off tracing for the named functions.
`(help)`	On-line help. `help` can take an argument, e.g., a function name.
`(vi filename)` [F]	Invoke the vi editor on the file `filename`.
`(vil filename)` [F]	Same as `vi`, except that `load` is executed on the file after vi is exited.

10 More hints

In order to invoke the emacs editor instead of vi in FRANZ LISP, use the following functions `em` and `eml` instead of `vi` and `vil`, respectively:

```
    (defun em fexpr (x) (exvi 'emacs x nil))
    (defun eml fexpr (x) (exvi 'emacs x t))
```

To save the output of your program into a file, you can use the UNIX command `script`. To use it, first execute `script`, then produce the output you want to save, and, finally, execute `exit` in the shell. At that point, the output will be in the file `typescript`, which you can then edit and print.

The COMMON LISP functions `defun.fexpr` and `concat-symbols` that were mentioned above are not a built-in functions. Their definition follows. The functions `trace.fexpr` and `untrace.fexpr` are used to trace and untrace fexpr functions.

```
(defmacro defun.fexpr (funname varlist &rest expr)
  (let ((ffunname (concat-symbols funname '.fexpr)))
    `(progn
      (defun ,ffunname ,varlist ,@expr)
      (defmacro ,funname (&body args)
        (list ',ffunname (list 'quote args))))))

(defun fexprlist (el)
  (if (null el) nil
    (cons (concat-symbols (car el) '.fexpr)
          (fexprlist (cdr el)))))

(defmacro trace.fexpr (&body funname)
  (cons 'trace (fexprlist funname)))

(defmacro untrace.fexpr (&body funname)
  (cons 'untrace (fexprlist funname)))

(defun concat-symbols (sym1 sym2)
  (intern (concatenate 'string (string sym1) (string sym2))))
```