Documentation for LISP in BASIC

LISP in BASIC is a LISP interpreter for a Scheme-like dialect of LISP, which happens to have been written in BASIC.

1. Starting Up

To launch LISP-in-BASIC type its name at the command-line, or double click on its icon. A shell-window should come up; at the top of the window you should see the following message.

```
Initializing Memory...
Initializing Lisp Environment...
LISP in BASIC v1.3 by Arthur Nunes-Harwitt
0]
```

The bracket (‘]’) is the prompt. (Note that there is a number right before the bracket. This number is the number of open left parentheses.) Type a symbolic expression here. If it is a complete expression — no remaining open parentheses, then the expression is evaluated; otherwise, another prompt appears.

2. Quitting

To quit, type the expression `(quit)` at the prompt. Evaluating that expression in a procedure also stops the interpreter.

3. The Language

3.1 A Brief Introduction to LISP

In many programming languages, a program is a sequence of state altering operations. One can program that way in LISP, but a more mathematical style is possible and preferable. In math, an important way to create new functions is to compose existing functions. For example, if f and g are existing functions then one can define the new function $h = f \circ g$ (or $h(x) = f(g(x))$). So too, in LISP one defines new procedures by composing existing procedures and syntactic forms. LISP syntax is a little different from mathematical syntax. Instead of writing *h*(*x*) = ..., one writes (define h (lambda (x) ...)); and instead of writing *f*(*g*(*x*)), one writes (f (g x)). A program is simply a collection of defined constants and procedures.

The name LISP stands for LISt Processing. The name is due to the fact that the list is the primary data structure in LISP. A list is a sequence of objects; an object can be a symbol, a number, another list, and even a procedure. The syntax of a list is simply the sequence of its objects separated by spaces and surrounded by parentheses. Interestingly, the syntax of a list is similar to the syntax of a procedure call. In fact, to prevent such a structure from being evaluated when typed in, the structure must be quoted.

There are a small number of basic list operations. Once these are mastered, one can do just about anything with lists. The procedure car returns the first element of the list. The procedure cdr returns the rest of the list (after the first element). The procedure cons puts those pieces back together to form a list.

Example:

```
(define L '(1 2 3)) ⇒ L
(car L) ⇒ 1
(cdr L) ⇒ (2 3)
(cons (car L) (cdr L)) ⇒ (1 2 3)
(define second (lambda (LST) (car (cdr LST)))) ⇒ second
(second L) ⇒ 2
```

Another important procedure is null?. It determines whether or not a list is empty. (Sometimes the empty list is referred to as nil. In fact, the variable NIL is initially bound to the empty list in LISP-in-BASIC.)

Example:

```
(null? L) ⇒ ()
(null? '()) ⇒ T
```

In LISP special iteration constructs are not needed. Recursive definitions are more powerful and more elegant. Suppose we wanted to compute the last element of a list. If a list has many elements, then clearly the last of the rest of the list is the same as the last of the list. If a list has only one element, then that element is the last element. That reasoning is sufficient for the purpose of writing a procedure.

Example:
```
(define last
  (lambda (S)
    (if (null? (cdr S))   If the rest of the list is empty, e.g. it has only one element
        (car S) then the answer is the first element, which is also the last element
        (last (cdr S))))) otherwise the answer is the last of the rest of the list
⇒ last
(last L) ⇒ 3
```

3.2 Primitive Types

The following classes of objects exist in LISP-in-BASIC.

- number (e.g. 123 and 3.14)
- symbol (e.g. foo and bar)
- pair (e.g. (*x* . *y*))
- primitive (e.g. the value of the symbol car in the initial environment)
- procedure (user defined procedures)

Note, though, that there is no primitive procedure to distinguish primitives from procedures. Both cause procedure? to evaluate to T.

There is no boolean class. The empty list is treated as the boolean false. There is a boolean true, though. In the initial environment, the symbol T is bound to true, and NIL is bound to the empty list. The empty list is printed as ().

3.3 Syntactic forms

and

Form:

(and exp_1 exp_2 … exp_n)

Examples:
(and (= 1 1) (car '(1 2 3))) ⇒ 1
(and nil (= 1 1)) ⇒ ()

Explanation:
And continues to evaluate its arguments until it hits an expression that evaluates to the empty list. If no expressions evaluate to the empty list, then it returns the value of the last expression.

cond

Form:
(cond (*test*$_1$ *exp*$_{1,1}$ ... *exp*$_{1,j}$) ... (*test*$_n$ *exp*$_{n,1}$ ... *exp*$_{n,k}$))
where *test*$_n$ may be simply the symbol else

Examples:
(let ((x '(1 2 3))) (cond ((null? x) nil) ((= (car x) 2) x) (else t)))
⇒ T
(let ((x '(2 3))) (cond ((null? x) nil) ((= (car x) 2) x) (else t)))
⇒ (2 3)

Explanation:
For each clause in the sequence, cond continues to evaluate the test expression of the clause until one evaluates to a non-nil value. Upon reaching such a clause, the expressions following the test are evaluated, and the cond clause evaluates to the last expression in that clause. (The symbol else is treated specially in the cond expression. If it is the last clause, and it is reached, the remaining expressions in that clause are evaluated.) Cond comes in handy when defining recursive procedures, or other procedures that need to make a decision.

define

Form:
(define *sym exp*)

Examples:
(define five 5) ⇒ FIVE

```
five ⇒ 5
(define short-alpha '(a b c d)) ⇒ SHORT-ALPHA
short-alpha ⇒ (A B C D)
(define double (lambda (n) (* 2 n))) ⇒ DOUBLE
(double 3) ⇒ 6
```

Explanation:
Define is used to bind a variable to a value. A define expression evaluates to the name of the variable being defined. (A value can be a procedure.)

if

Form:
(if *test* exp_1 exp_2) or (if *test exp*)

Examples:

```
(if (= 1 1) 3 4) ⇒ 3
(if (= 1 2) 3 4) ⇒ 4
(if (= 1 2) 3) ⇒ ()
```

Explanation:
If, like cond, is used for making decisions. However, if has a much simpler structure. If the test evaluates to the empty list then the expression evaluates to what exp2 evaluates to. Otherwise, the expression evaluates to what exp1 evaluates to. (It is possible to leave off the second expression, in which case the second expression is implicitly the empty list.)

lambda

Form:
(lambda (var_1 … var_m) exp_1 … exp_n)

Examples:

```
(define double (lambda (n) (* 2 n))) ⇒ DOUBLE
(double 3) ⇒ 6
(define make-adder (lambda (n) (lambda (m) (+ n m))))
```

⇒ MAKE-ADDER
(define add3 (make-adder 3)) ⇒ ADD3
(add3 7) ⇒ 10

Explanation:
A lambda expression evaluates to a procedure. This procedure can then be applied — as in the examples. Writing a LISP program consists of writing such procedures.

let

Form:
(let ((*var*$_1$ *exp*$_1$) ... (*var*$_m$ *exp*$_m$)) *exp*$_1$' ... *exp*$_n$')

Example:
(let ((x 5)) (+ x 1)) ⇒ 6

Explanation:
Local variables are defined with a let expression. Note that the variables are bound 'in parallel' so none of the let variables can be defined in terms of the others. The let expression evaluates to the last expression in the implicit sequence after the bindings.

let*

Form:
(let* ((*var*$_1$ *exp*$_1$) ... (*var*$_m$ *exp*$_m$)) *exp*$_1$' ... *exp*$_n$')

Example:
(let* ((x 2) (y (+ 1 (* 2 x)))) (+ y 2)) ⇒ 7

Explanation:
Let* does almost the same thing as let, but with let*, the variables are bound in sequence so a variable can be defined in terms of those preceding it.

or

Form:

(or *exp*$_1$ *exp*$_2$ ... *exp*$_n$)

Examples:

(or (= 1 2) (car '(1 2 3)) (cdr '(1 2 3))) ⇒ 1

(or (= 1 3) (= 1 2)) ⇒ ()

Explanation:

Or continues to evaluate its arguments until it hits an expression that does not evaluate to the empty list. At that point, the expression evaluates to that value. If no non-nil values are found, the expression evaluates to the empty list.

quote

Form:

(quote *exp*) or '*exp*

Examples:

(quote (1 2 3)) ⇒ (1 2 3)

'(1 2 3) ⇒ (1 2 3)

'a ⇒ a

(+ '5 '6) ⇒ 11

Explanation:

Quote prevents its expression from being evaluated. If a list were not quoted, the expression would be evaluated, and a parenthesized expression would be treated as a procedure call.

sequence

Form:

(sequence *exp*$_1$ *exp*$_2$ ... *exp*$_n$)

Example:

(define temp 3) ⇒ TEMP

(sequence (set! temp 5) t) ⇒ T

Explanation:
Sequence evaluates a sequence of expressions in order from left to right. The expression evaluates to the last expression in the sequence. Sequences are used when expressions that perform side effects are involved.

set!

Form:
(set! *var exp*)

Example:
```
(define temp 3) ⇒ TEMP
temp ⇒ 3
(set! temp 5) ⇒ 5
temp ⇒ 5
```

Explanation:
If a variable is bound to a value, set! changes the value associated with the variable. The new value is what the second argument evaluates to. The set! expression evaluates to the new value.

3.4 Built-in Procedures

*

Examples:
```
(*) ⇒1
(* 2) ⇒2
(* 2 3) ⇒6
(* 2 3 4) ⇒24
```

Explanation:
The * procedure is the multiplication or product procedure.

+

Examples:
```
(+) ⇒ 0
(+ 2) ⇒ 2
```

(+ 2 3) ⇒ 5
(+ 2 3 4) ⇒ 9

Explanation:
The + procedure is the addition or sum procedure.

-

Examples:
(- 2) ⇒ -2
(- 3 2) ⇒ 1
(- 3 2 1) ⇒ 0

Explanation:
When called with one argument the - procedure performs negation. When called with more than one argument, the - procedure subtracts from the first argument the rest of the arguments.

/

Examples:
(/ 2) ⇒ .5
(/ 12 2) ⇒ 6
(/ 12 2 3) ⇒ 2

Explanation:
When called with one argument the / procedure computes the reciprocal. When called with more than one argument, the / procedure divides the first argument by the rest of the arguments.

<

Examples:
(< 2 3) ⇒ T
(< 2 2) ⇒ ()
(< 2 1) ⇒ ()

Explanation:
The < procedure is the less-than procedure. If the first argument is less than the second it evaluates to true, otherwise it evaluates to nil.

<=

Examples:
(<= 2 3) ⇒ T

(<= 2 2) ⇒ T
(<= 2 1) ⇒ ()

Explanation:
The <= procedure is the less-than-or-equal procedure. If the first argument is less than or equal to the second it evaluates to true, otherwise it evaluates to nil.

=

Examples:
(= 2 3) ⇒ ()
(= 2 2) ⇒ T
(= 2 1) ⇒ ()

Explanation:
The = procedure is the numeric equal procedure. If the first argument is numerically equal to the second it evaluates to true, otherwise it evaluates to nil.

>

Examples:
(> 2 3) ⇒ ()
(> 2 2) ⇒ ()
(> 2 1) ⇒ T

Explanation:
The > procedure is the greater-than procedure. If the first argument is greater than the second it evaluates to true, otherwise it evaluates to nil.

>=

Examples:
(>= 2 3) ⇒ ()
(>= 2 2) ⇒ T
(>= 2 1) ⇒ T

Explanation:

The > procedure is the greater-than-or-equal procedure. If the first argument is greater than or equal to the second it evaluates to true, otherwise it evaluates to nil.

apply

Examples:

```
(apply + '(1 2 3)) ⇒ 6
(apply cons '(1 2)) ⇒ (1 . 2)
```

Explanation:
The apply procedure takes two arguments: a function and a list. The function is then applied to the elements of the list as the function's arguments.

assoc

Examples:

```
(assoc '(a b c) '(((x y) z) ((a b c) d))) ⇒ ((A B C) D)
(assoc 'z '((x y) (a b) (c d))) ⇒ ()
```

Explanation:
The procedure assoc searches an association list. It returns the first pair whose first element is equal? to the first argument. If there is no such pair, then nil is returned.

assq

Examples:

```
(assq '(a b c) '(((x y) z) ((a b c) d))) ⇒ ()
(assq 'a '((x y) (a b) (c d))) ⇒ (A B)
(assq 'z '((x y) (a b) (c d))) ⇒ ()
```

Explanation:
The procedure assq searches an association list. It returns the first pair whose first element is eq? to the first argument. If there is no such pair, then nil is returned.

atan

Examples:

```
(atan 1) ⇒ .7853981
(atan 0) ⇒ 0
```

Explanation:
The procedure atan computes the arc-tangent function. (Angles are measured in radians.)

bound?

Example:

(define temp 5) ⇒ TEMP

(bound? 'temp) ⇒ T

Explanation:
The procedure bound? returns true if its argument is a symbol that is bound in the current environment, otherwise it returns nil.

car

Examples:

(car '(1 2 3)) ⇒ 1

(car '(x . y)) ⇒ X

Explanation:
The procedure car returns the first element of a pair.

cdr

Examples:

(cdr '(1 2 3)) ⇒ (2 3)

(cdr '(x . y)) ⇒ Y

Explanation:
The procedure cdr returns the second element of a pair. When applied to a proper list, the result is often called the but-first of the list or the rest of the list.

cons

Examples:

(cons 'x 'y) ⇒ (X . Y)

(cons 1 '(2 3)) ⇒ (1 2 3)

Explanation:
The procedure cons constructs a pair. When the second argument is a list, the first argument appears to become the new first element of that list.

cos

Examples:

(cos .7853981) ⇒ .707168

(cos 1.570796) ⇒ 7E-08

(cos 1.047197) ⇒ .5000001

Explanation:
The procedure cos computes the cosine function. (Angles are measured in radians.)

eq?

    Examples:
    (eq? 'a 'a) ⇒ T
    (eq? 'a 'b) ⇒ ()
    (eq? '(1 2) '(1 2)) ⇒ ()

Explanation:
The procedure eq? returns true if its arguments are identical, otherwise it returns nil. (Symbols are always identical; however, lists may look similar, but not be identical.)

equal?

    Examples:
    (equal? 'a 'a) ⇒ T
    (equal? 'a 'b) ⇒ ()
    (equal? '(1 2) '(1 2)) ⇒ T
    (equal? '(1 2) '(3 4)) ⇒ ()

Explanation:
The procedure equal? returns true if its arguments have the same structure, otherwise it returns nil.

eval

    Example:
    '(+ 2 3) ⇒ (+ 2 3)
    (eval '(+ 2 3)) ⇒ 5

Explanation:
The procedure eval evaluates a list structure as if it were LISP code.

exp

    Examples:
    (exp 1) ⇒ 2.718282
    (exp 2) ⇒ 7.389056

(exp (log 3)) ⇒ 3

Explanation:
The procedure exp computes the exponential function (i.e. e^x).

floor

Examples:

```
(floor 1.6) ⇒ 1
(floor -1.6) ⇒ -2
```

Explanation:
The procedure floor computes the floor function.

for-each

Example:

```
(define temp (list (cons 'a 1) (cons 'b 2) (cons 'c 3))) ⇒ TEMP
(for-each set-cdr! temp '(7 8 9)) ⇒ ()
temp ⇒ ((A . 7) (B . 8) (C . 9))
```

Explanation:
The procedure for-each is similar to map, except that for-each is used for its side effects; it always evaluates to nil. It applies its first argument, which should be a function, to the corresponding elements of the rest of its arguments, which should be lists, in sequence.

gc

Example:

```
(gc)
```

Explanation:
The procedure gc causes LISP-in-BASIC to perform a garbage collection. It returns the number of memory cells available.

length

Example:

```
(length '(a b c)) ⇒ 3
```

Explanation:
The procedure length computes the length of a list.

list

  Examples:

  (list 1 2 3) ⇒ (1 2 3)

  (list '(a) '(b)) ⇒ ((A) (B))

Explanation:
The procedure list constructs a list of its arguments and returns that list.

load
Explanation:
The procedure load takes one argument: a symbol that is the file name of the file to be loaded. (The file should be a text file.) The file is assumed to be a sequence of LISP expressions. They are read in one at a time and evaluated until there are no more.

log

  Examples:

  (log 2.718281828) ⇒ 1

  (log 10) ⇒ 2.302585

Explanation:
The procedure log computes the natural logarithm function (i.e. ln(*x*)).

map

  Examples:

  (map (lambda (x) (* x x)) '(1 2 3)) ⇒ (1 4 9)

  (map cons '(a b c) '(1 2 3)) ⇒ ((A . 1) (B . 2) (C . 3))

Explanation:
The procedure map returns a new list whose elements are the values computed by applying the first argument, a function, to the corresponding elements of the rest of the arguments.

member

  Examples:

  (member '(a b) '((x y) (a b) (c d))) ⇒ ((A B) (C D))

  (member '(e f) '((x y) (a b) (c d))) ⇒ ()

  (member 'c '(a b c d e f)) ⇒ (C D E F)

  (member 'g '(a b c d e f)) ⇒ ()

Explanation:
The procedure member searches the second argument, which must be a list, for the first argument. If it finds an element equal? to the first argument, it returns the rest of the list starting with that element. Otherwise, it returns nil.

memq

Examples:

```
(memq '(a b) '((x y) (a b) (c d))) ⇒ ()
(memq 'c '(a b c d e f)) ⇒ (C D E F)
(memq 'g '(a b c d e f)) ⇒ ()
```

Explanation:
The procedure memq searches the second argument, which must be a list, for the first argument. If it finds an element eq? to the first argument, it returns the rest of the list starting with that element. Otherwise, it returns nil.

newline
Explanation:
The procedure newline has the side effect of causing the display to move down one line. This procedure always evaluates to T; it takes no arguments.

not

Examples:

```
(not (= 1 1)) ⇒ ()
(not (= 1 2)) ⇒ T
```

Explanation:
The procedure not performs logical negation. If its argument is nil, it returns true. Otherwise, it returns nil.

null?

Examples:

```
(null? (cdr '(1))) ⇒ T
(null? '(1)) ⇒ ()
```

Explanation:
The procedure null? is a predicate that determines whether its argument is the empty list; if so, it returns true, otherwise it returns nil.

number?

Examples:

```
(number? 2) ⇒ T
(number? 'a) ⇒ ()
(number? '(1 2)) ⇒ ()
```

Explanation:
The procedure number? is a predicate that determines whether its argument is a number; if so, it returns true, otherwise it returns nil.

pair?

Examples:

```
(pair? '(1 . 2)) ⇒ T
(pair? '(1 2)) ⇒ T
(pair? 'a) ⇒ ()
(pair? 2) ⇒ ()
```

Explanation:
The procedure pair? is a predicate that determines whether the its argument is a pair; if so, it returns true, otherwise it returns nil.

print
Explanation:
The procedure print has the side effect of displaying on the screen its argument. It also returns the value of its argument.

procedure?

Examples:

```
(procedure? car) ⇒ T
(procedure? (lambda (x) x)) ⇒ T
(procedure? 'car) ⇒ ()
(procedure? '(lambda (x) x)) ⇒ ()
(procedure? 2) ⇒ ()
```

Explanation:
The procedure procedure? is a predicate that determines whether its argument is a procedure; if so, it returns true, otherwise it returns nil.

quit
Explanation:
The procedure quit causes the LISP-in-BASIC interpreter to quit. Unlike all the other procedures, this one does not (cannot) return a value.

read
Explanation:
The procedure read takes no arguments. When read is called, the user may type in a symbolic expression; this expression is then returned.

reverse

    Example:
    (reverse '(1 2 3 4)) ⇒ (4 3 2 1)

Explanation:
The procedure reverse must have a list as its argument. It then reverses that list, and returns the reversed list.

set-car!

    Example:
    (define temp '(1 2 3)) ⇒ TEMP
    (set-car! temp 'a) ⇒ a
    temp ⇒ (a 2 3)

Explanation:
The procedure set-car! destructively modifies the first element of a pair. The first argument has its car reset to be the second argument. It returns simply the value of its second argument.

set-cdr!

    Example:
    (define temp '(a . b)) ⇒ TEMP
    (set-cdr! temp 2) ⇒ 2
    temp ⇒ (a . 2)

Explanation:
The procedure set-cdr! destructively modifies the second element of a pair. The first argument has its cdr reset to be the second argument. It returns simply the value of its second argument.

sin

    Examples:
    (sin .7853981) ⇒ .7071068
    (sin 1.570796) ⇒ 1
    (sin 1.047197) ⇒ .8660254

Explanation:
The procedure sin computes the sine function. (Angles are measured in radians.)

symbol?

Examples:

```
(symbol? 'a) ⇒ T
(symbol? T) ⇒ ()
(symbol? nil) ⇒ ()
(symbol? '(a b c)) ⇒ ()
(symbol? 5) ⇒ ()
```

Explanation:
The procedure symbol? is a predicate that determines whether its argument is a symbol; if so, it returns true, otherwise it returns nil.