Micro-Lisp Version 2.5

A public domain Lisp interpreter for the C64!

by Nicholas Vrtis

Lisp is a language designed to work with lists (its name is a contraction of LISt Processor). It is one of the primary languages used in the study of Artificial Intelligence. Micro-Lisp is a subset of this language that you can use to learn more about its capabilities. Although there obviously isn't space in this article for a complete course in Lisp and AI, I would like to introduce you to the language and to my Micro-Lisp implementation in particular. At the end of the article you'll find some suggestions for further reading if you want to know more.

Why bother with a version of Lisp that runs on a slow 8 bit computer? Because it is an easy, inexpensive way to become familiar with the language, and to get a feel for what it is like. Why buy a model rocket? It can't go as fast or as far as the Space Shuttle, but you can still have fun and learn from it.

Lisp's World View

Lisp divides the world into two classes of 'things'. On one side there is a *List*, and on the other is things that aren't Lists. Things that aren't Lists are called *Atoms*. As in physics, an Atom can't be broken down into something smaller (though you will find that you can *explode* and *implode* an Atom, just as in physics atoms can be taken apart if you know how). A word is an Atom, so is a number. If I take some Atoms, and collect them, I end up with a List (similar to taking atoms and collecting them into molecules). I can also take Lists and group them together, either end-to-end to make a longer List, or as a List of Lists.

Big deal - what good are lists anyway? Actually, if you think about it, almost all the information we use is in the form of lists. Your checkbook for example, is a list of three or four items of information about each check (the check number, the amount, who it is to, the date written, and possibly a budget category). To get the amount I have spent on a given category, I just go through my list of checks and add up the amounts for those checks with that category. That's a relatively simple Lisp application, and maybe not a very appropriate one - a good database program would probably be better, since it would allow you to sort the data and print a fancy report without a lot of work.

Lisp was designed to handle more complicated situations where you can't know in advance all the combinations and questions you might want to ask about the information you have. One example (which I'll be using throughout this article) would be a family tree. A List showing the name, sex, and parents of each individual in your family would be a good starting point. Each item on this List is made up of two Atoms (name and sex) and one List (Father's name, Mother's name). Notice that there is no item in the List concerning the individual's relationship to you, or to most other members of the family. We can get this information, however, by applying some simple rules; for instance: *a brother of x is any male whose parents are the same as the parents of x.* Before we discuss how Lisp lets us extract this kind of implicit information, however, we need to master a few of the language's technicalities.

Lisp Fundamentals

Lists in Lisp are enclosed in parentheses. For instance, while *nick* is an Atom, *(nick)* is a List that has one Atom, *(nick m)* has two, and *(nick m (jim marion))* has two Atoms and a List (which itself has two Atoms). By the way, one of the hard parts about Lisp, especially for beginners, is keeping the parentheses balanced in the right places. Micro-Lisp has a couple of features to help with this. The command prompt shows the current number of unbalanced parentheses (the *nesting* level). Also, when you display a List, you can use a feature called *pretty print* to start each new level on a new line, and indent one space for each level.

Another concept you need to understand about Lisp is how it represents 'nothing'. Since an Atom can be either a number or a word, Lisp can't use 0 for numbers the way we do. Instead, Lisp uses an entity called *nil*. *Nil* is special, because it can be either a List or an Atom depending on the situation. If you want to input the Atom *nil*, just enter the word *nil*. If you want to input a List with nothing in it, enter *()*. Whenever Micro-Lisp displays an empty List, it will always display *nil* instead of *()*.

We also need to understand how to get Lisp to do something - how to give it a command. Commands are given in the form of Lists (not surprisingly). A command List is no different from any other List, except that the first entry must be an Atom that is a command. For example, *add* is a command that sums the numbers in the rest of the List; the List *(add 1 2 3)* would add the numbers 1, 2 and 3. A List doesn't have to have a command as the first entry unless you want to execute it (called *evaluating* it in Lisp). The documentation accompanying this article shows the built-in commands available in Micro-Lisp. One of these - *define* - lets you create your own commands, which work just like the built-in commands. We'll make use of this ability when we work on our family-tree project.

Creating the family tree

Let's begin that now. To start Micro-Lisp, just enter:

```
load "micro-lisp",8
run
```

You'll see a title message and a flashing cursor. Now type:

```
>0 (set (quote family-tree) nil)
nil
```

The computer will respond by typing out *nil* (note: the examples in this article use **bold** type for the computer's prompts and responses, and regular type for your input). *Set* is a command that sets the value of the second Atom in the List equal to the value of the third (similar to a BASIC statement like FT$ = ""). Now we have a 'database' named FAMILY-TREE with nothing in it.

You might be wondering why you had to use the strange construction *(quote family-tree)* instead of just *family-tree*. This reflects LISP's desire to use the *value* of a name in most cases. Consider the BASIC statement FT$ = A$, which assigns the value of A$ to FT$. If we really wanted to assign the characters "A$" to FT$, we have to use quotes. The *quote* command in LISP performs the same function as the pair of double quotes in BASIC. If that is still confusing, try this:

```
0> (set (quote tree-name) (quote family-tree))
family-tree
0> (set tree-name 10)
10
0> tree-name
family-tree
0> family-tree
10
```

In this example, we begin by creating a new Atom - *tree-name* - whose value is the name *family-tree*. When we now say *(set tree-name 10)*, we are asking for the value 10 to be assigned to the Atom whose name is found by evaluating *tree-name*. After this operation, we discover that the value of *tree-name* is unchanged but, as expected, *family-tree* has the new value 10. Programmers who have used languages like assembler, C and PROMAL will recognize here an example of *indirection*; this application of it is fundamental to Lisp and you should make sure you understand the above example thoroughly.

Since you end up using the *quote* command a lot in Lisp, there is a shorthand version. It is a single quote mark ('). It eliminates the word *quote* and a set of parentheses. We can thus write our original statement more concisely as:

```
(set 'family-tree nil)
```

Go ahead and try it - Lisp is very interactive. If you ever want to know the value of a name, just type it on the command line without parentheses.

A new command with *define*

Now let's define a command of our own to add a person to our database. We use the *define* command for this, and we'll keep it simple for now. Later you will probably want to add some checks to this command to guard against duplicate entries and to determine if the parents of a newly-added person are already defined. But start with:

```
0> (define 'add-person '(person)
1>     '(progn
3>         (setq family-tree
4>             (cons person family-tree))
3>         person))
```

Micro-Lisp should respond with *add-person*. If not, make sure you haven't missed any quotes or parentheses (hint: if you have a number greater than zero in front of the > prompt, that is the number of parentheses you are missing). You don't have to indent as shown above, though it helps show each level of the definition.

There are a number of new items in this definition, but most are pretty simple. *Define* is the command that defines new commands to Lisp. It needs to know three things. The first is the name of the command (we are calling it *add-person*); the second is a List of the arguments (only one in this case - *person*); and the last is the body of the function. Note that quote marks are used, since we want the literal statements we typed in, not their value.

The first command in the function body is *progn*. All this command does is tell Lisp to evaluate all the other items in the List. Normally, Lisp expects a command to be in the form (Command Argument Argument...). *Progn* lets you string a set of commands into one List in the form *(progn (Command Argument ...) (Command Argument ...) ...)*. We need to do this in *add-person* because we want to do two things. The first is *setq*. This is a special version of *set*, which we used earlier. *Setq* allows you to skip the first quote. The variable we are setting is *family-tree*, our database. What we want to set it to is a List consisting of all the things already in *family-tree* plus the new *person* data. We use the *cons* command to do this. *Cons* creates a List formed from the first argument followed by the second argument. Note that I put *person* first, and *family-tree* second, thus adding the new information to the front of the database instead of the end. For technical reasons this turns out to be faster than the other way around, but either way will work.

The second item is not a command, just the word *person*. Notice that it is not enclosed in parentheses. When Lisp sees just a variable name without parentheses, it just takes the value of that variable, and leaves it as the *return value*. Everything in Lisp leaves some sort of return value; whenever Lisp has finished processing the commands you have given it, it prints out the final return value. Well, it turns out the return value from *setq* is the value to which the variable was set. In our case, this is the whole database. Since it could get lengthy to have it print out every time we add a new person to *family-tree*, we add the word *person* by itself; now our new command has as its return value the information about the person we just added. We could have used a special variable called *t*, or *nil*, but *person* might be more useful if we want to combine *add-person* into some other command.

Let's test out what we have so far. Issue the following:

```
0>(add-person '(nick m (jim marion)))
(nick m (jim marion))
0>family-tree
((nick m (jim marion)))
```

The last should produce a List of Lists showing everything in our database. Since it is all scrunched together, enter *(setpretty t)*, then *family-tree* again. This will indent each level of parentheses and make the Lists a little easier to read. Now build up the database a little by adding some more people with the following lines (this time, Lisp's responses are not shown):

```
(add-person '(maryelna f (frank dorothy)))
(add-person '(nikki f (nick maryelna)))
(add-person '(mike m (jim marion)))
(add-person '(matt m (nick maryelna)))
family-tree
```

Notice that the database has become larger.

Interrogating the database

Now let's define some new commands that will help us find things in the database. The first is a command to find somebody's name and parentage:

```
0>(defun find-name (name)
1>    (setq temp family-tree)
1>    (dountil (or
3>              (eq temp nil)
3>              (eq name (car (car temp))))
2>          (setq temp (cdr temp)))
1>    (setq person (car temp)))
```

There are more new commands here, but it is still pretty simple. *Defun* is another version of *define*. Like *setq*, it is a shorthand that eliminates the need for quoting its arguments. *Defun* also supplies an implicit *progn*, so we don't have to bother with that either. We've seen *setq* before. Here we are using it to set a temporary variable to our database because we are going to have to check each item in it to see if the first Atom of the sublist is the name we are looking for.

Dountil is a looping command. Until the first argument returns a true value (in Lisp, *true* means anything that isn't *nil*), this command will execute the remaining commands in the List. In our case, we will want to terminate the loop when either we have run out of person entries in the database, or we have found the person entry whose first Atom is the name of the person we are looking for.

Conveniently, Lisp has an *or* command to express this sort of requirement. *Or* evaluates each of its arguments until it either finds a non-*nil*, or runs out of arguments (in which case it returns *nil*). The first argument is *(eq temp nil)*. The *eq* tests to see if its two arguments are identical, so if *temp* is equal to *nil*, this command will return *t*. If *temp* is not *nil*, *or* goes to the next argument. This means there is a person List left, so we want to compare the first Atom in that List with the name we are looking for.

To do this, we use the *car* command. *Car* returns the first part of its argument (which must be a List). Since *temp* is a copy of *family-tree*, the *car* (first item) of *temp* is the first 'person' List in *temp*. The name is the first Atom in the List, so we take the *car* of the *car* of *temp*, and compare that to the name we're looking for. If they are the same, the *eq* will return *t*, and the *dountil* is done. Otherwise we need to do something to look at the rest of the person Lists in *temp*.

This is where *cdr* comes in. *Cdr* returns the tail of a List - everything but the *car*. Since *or* told us that the current first List in *temp* isn't the one we want, we simply *setq temp* to everything but the first List, and repeat the process. Eventually, the *dountil* either runs out of Lists in *temp* (*temp* equals *nil*), or the *car* of *temp* is the person *list* corresponding to the name we want. When the *dountil* terminates, we return the *car* of *temp*. Note that the *car* of *()* is *nil*, so *find-name* returns either the person List of the name we asked for, or *nil* if the name is not found. Try *(find-name 'nikki)*; you should get back *(nikki f (nick maryelna))*.

Now for another simple, but useful command:

```
0>(defun parents-of (name)
1>    (cond
2>          ((find-name name) (setq parents (nth 3 person)))
2>      (t (setq parents nil))))
parents-of
```

Pretty easy, right? Only two new commands this time. *Cond* is the Lisp version of *if*, but a little more complicated. Basically, the arguments for *Cond* are the members of a List of *if* statements. Each argument List is a pair of commands. The first command in the pair is the condition part. If it returns non-*nil*, then the second command is evaluated. If the first command returns *nil*, then the next condition is examined. There is a requirement in Micro-Lisp that at least one of the conditions in a *cond* statement must return non-*nil*, or it is considered an error.

In *parents-of*, the first condition is *(find-name name)*. Recall that *find-name* returns the person List entry if the name is found, or *nil* if it is not. If *(find-name name)* returns non-*nil* in the present case, we will want to set a variable called *parents* to the third item in the person List that *find-name* returned. To do this we use *nth*, a command that returns the nth entry from the third argument (a List), where *n* is specified by the second argument.

In case the first condition returns *nil* (the name was not found), we need to make sure that at least one condition is true (non-*nil*). Lisp supplies a variable called *t* that is guaranteed to return non-*nil*; we use this for the second condition, and set *parents* to *nil* because we can't identify the parents of someone not in the database. Note that Lisp skips condition testing after the first true condition is found, so the second *setq* in the above definition is never executed if the name we are looking for is found. Try *(parents-of 'nick)*; you should get back *(jim marion)*.

One more short example:

```
0> (defun grand-parents-of (name)
1>    (setq grand-parents
2>       (list
3>          (parents-of (car (parents-of name)))
4>             (parents-of (car (cdr (parents-of name)))))))
```

Only one new command in the whole thing, and it is pretty easy to figure out what it does. *List* takes all its arguments and returns a List (simple, isn't it?). Think about what is going on. Grandparents are parents of a person's parents, so all our new command has to do is create a List of the parents of the parents of the person in *name*. *Parents-of* returns a List with the two parents' names. The *car* (first part) of this List is the name of one of the parents. If we now call *parents-of* with this, we will get one set of grandparents. The *cdr* (rest of) the original parent List is a List (*cdr* always returns a List) that has only one entry, the other parent. The *car* of that is the name of the other parent, and the *parents-of* that is the other set of grandparents. It takes a long time to explain, but it really isn't complicated - just follow it through. Try *(grand-parents-of 'matt)*; you should get back *((jim marion) (frank dorothy))*.

Where to go from here

I could continue with more examples, but these should give you an idea of what Lisp is. Purists will probably be upset that I did not use recursion techniques in the examples. Lisp handles these very well, but I find them difficult to follow and harder to explain; I purposely kept the examples straightforward. As you can see by examining the list of built-in commands, there are a lot of Lisp words that I didn't even cover. Experiment with them. Even more than BASIC, Lisp is interactive. Try some things and see what happens.

After you have some experience, try the command called *setdebug*. This turns on a trace facility that traces what is going on. Then use *baktrack* to see all that went on to get to where you are (with our simple *family-tree*, *grand-parents-of* goes through over 100 Lisp statements to get the answer). There is also a *trace* command that prints out the levels as they are being executed. If *setdebug* is *t* and there are symbolic variables (as *name* was in our examples) you get an opportunity to display their values before Lisp restores them (enter *nil* to get out of this mode).

There is an editor (entered with the *edit* command) that allows you to input and save Micro-Lisp source statements; you use the Micro-Lisp command *source* to load them into your Lisp 'environment'. To Micro-Lisp, your 'environment' is all the Lists and Atoms you have defined. Use the commands *save* and *load* to keep and restore copies of your work.

A separate program ("sae.lisp") is a special version of the Micro-Lisp *edit* command. This program runs without Micro-Lisp to let you create Micro-Lisp source programs larger than would be possible with Micro-Lisp running (since Micro-Lisp and all your working Lists take up memory).

Meanwhile, if Lisp interests you enough that you would like to know more about it, you might want to read some or all of these books and articles:

Programmer's Guide To Lisp, by Ken Traction (Tab Books)
Understanding Lisp, by Paul Gloess (Alfred Publishing)
Lisp: Basically Speaking (*80 Micro*, May 1983)
Design Of AN M6800 Lisp Interpreter (*Byte*, August 1979)
Three Microcomputer Lisps (*Byte*, September 1981)

* * * * *

Editor's Note: *Unfortunately, with a 10K object file size, there is just no room for a program listing of Nick Vrtis' Micro-Lisp interpreter in the magazine. However, Micro-Lisp, some sample Lisp code and the SAE.lisp stand-alone editor will be available on the disk for this issue, and will also be posted to Data Library 17 on CompuServe's CBMPRG Forum. In the near future,* ***Transactor*** *will also be releasing a special disk containing the MAE assembler source to Micro-Lisp, along with programming notes for those who wish to expand or modify the interpreter for their own needs.*

Micro-Lisp Built-In Functions

ABORT **(ABORT msg)**

Stops current processing, displays the message that *msg* evaluates to, and returns to the top level processing. This function can be used to define additional error processing.

ABS **(ABS n)**

Returns the absolute value of *n*.

ADD **(ADD n1 ... nn)**

Returns the sum of the numbers *n1* through *nn*.

AND **(AND x1 ... xn)**

Evaluates *x1* and, if it is not *nil*, proceeds to evaluate the following arguments until *nil* is returned, or the end of the argument List is encountered. Note that *and* does not evaluate the following arguments if *nil* is returned from any argument.

APPEND **(APPEND l1 l2)**

Returns the List created by appending *l2* to the end of *l1*.

APURGE **(APURGE atm)**

Purges the Atom *atm* from memory and frees the space for reuse. Any references to *atm* in Lists are changed to reference *nil*. *Apurge* removes the value, definition, and property Lists of built-in functions, but does not remove the built-in definition or free the space.

ATOM **(ATOM x)**

Returns *t* (true) if *x* evaluates to an Atom; returns *nil* otherwise.

BAKTRACK **(BAKTRACK n)**

Displays the last *n* functions that were executed. Only valid if *debug* has been previously activated (see *setdebug*).

CAR **(CAR list)**

Returns the first part of *list*.

CDR **(CDR list)**

Returns the rest of *list* after the first part (the *car*) is skipped.

COLD **(COLD)**

Restarts Micro-Lisp from the beginning. All options and parameters are reset to the way they were when RUN was issued.

COND **(COND (t1 r1)(t2 r2))**

Evaluates *t1* and executes *r1* if *t1* returns non-*nil*. If *t1* returns *nil*, *cond* proceeds to evaluate *t2*. Note that *t2* will not be evaluated if *t1* does not evaluate to *nil*. An error is issued if all tests evaluate to *nil*.

CONS **(CONS x1 x2)**

Returns a List that has *x1* as its first part (*car*) and *x2* as its rest part (*cdr*). If *x1* and *x2* evaluate to Atoms, a special List called a 'Dotted Pair' is returned.

DECIMAL **(DECIMAL)**

Sets the current number base to decimal and returns the number 10.

DEFINE **(DEFINE fn arg x)**

Create a function called *fn* that has arguments specified in the list *arg*. The body of the function is specified in the list *x*. In *define*, *fn*, *arg*, and *x* are evaluated. A defined function has an implied *progn* before the body.

DEFUN **(DEFUN fn arg x)**

Same as *define*, except that *fn*, *arg*, and *x* are not evaluated. It is a more convenient form when entering a function from the keyboard since quotes are not needed.

DISKCMD **(DISKCMD msg)**

Issues the disk command *msg*. This function opens and closes the disk command channel.

DISKST **(DISKST)**

Reads the disk status via the command channel. The function opens and closes the command channel.

DISK$** **(DISK$)

Reads the disk directory and returns a List of Lists about the disk. Each entry is a List of 3 Atoms. The first Atom is the number of blocks, the second is the file name, and the third is the file type. The first entry is the disk name, and the last entry is the number of blocks free.

DIVIDE **(DIVIDE n1 n2)**

Returns the integer result of *n1* divided by *n2*.

DOUNTIL **(DOUNTIL t x1 ... xn)**

A looping function. The test *t* is evaluated and, until it is true (non-*nil*), the expressions *x1* through *xn* are evaluated (using an implied *progn*). *Dountil* returns the non-*nil* value returned from the expression *t*.

DOWHILE **(DOWHILE t x1 ... xn)**

A looping function. The test *t* is evaluated and, while it is still true (non-*nil*), the expressions *x1* through *xn* are evaluated (using an implied *progn*). *Dowhile* returns *nil* always.

EDIT **(EDIT)**

Exits Lisp to the BASIC screen editor. You may enter a Lisp program as you would a BASIC program. The only keywords that will function are LOAD, SAVE, LIST, NEW and END. END returns you back to Micro-Lisp, the others operate as expected.

EQ **(EQ x1 x2)**

Returns *t* if *x1* is identical to *x2*. Not very useful if comparing Lists.

EQUAL **(EQUAL x1 x2)**

Returns *t* if *x1* is the same as *x2*, otherwise returns *nil*. This will properly test Lists.

EVAL **(EVAL x)**

Evaluates the List that *x* evaluates to. *(Eval '(car '(a b)))* will produce the same results as *(car '(a b))*.

EXIT **(EXIT)**

Leaves Lisp and returns to BASIC.

EXPLODE **(EXPLODE a)**

Returns a List of numbers that represent the ASCII value of the name of the Atom *a*. If *a* evaluates to a number, it is converted according to the current base and the ASCII values of the digits are returned.

GC **(GC)**

Forces garbage collection to take place. Returns *t*.

GETDEF **(GETDEF a)**

Returns the definition of the function specified by *a*. Returns *nil* if *a* has not been previously *define*d.

GETPROP **(GETPROP a pn)**

Returns the property value stored under the property name *pn* under the atom *a*. Returns *nil* if *pn* is not defined under *a*.

GREATERP **(GREATERP n1 ... nn)**

Returns *t* if the numbers *n1* through *nn* are in descending order. Note that equal is not considered descending.

HEX **(HEX)**

Set the current number conversion base to 16 (hexadecimal). Returns the value 10 hex (decimal 16).

IMPLODE **(IMPLODE l)**

Takes a List of numbers representing ASCII characters and returns an Atom whose print name is that series of ASCII characters. If any number is greater than 256, the ASCII character is taken from the MOD 256 value. If the print name resulting from imploding the List results in a valid number, then it will be converted to a number.

LAMBDA **((LAMBDA (x) (b)) y)**

A method of executing a function without defining it. The current value of *x* is saved, then it is assigned the value of *y* and the body *b* is evaluated (with an implied *progn*). After *b* is evaluated, *x* is restored to its previous value. Note that there may be more than one Atom specified in the argument list, and that there must be a one to one correspondence between the number of arguments and the number of values supplied.

LENGTH **(LENGTH x)**

Returns the number of elements in the list *x*.

LESSP **(LESSP n1 ... nn)**

Returns *t* if the numbers *n1* through *nn* are in ascending order. Note that equal is not considered ascending.

LIST **(LIST x1 ... xn)**

Returns a List containing *x1* through *xn*.

LISTP **(LISTP x)**

Returns *t* if *x* evaluates to a List, *nil* if *x* evaluates to an Atom.

LOAD **(LOAD fn)**

Loads a previously saved environment from file *fn*. This must be executed from the first level.

LPAREN **(LPAREN)**

Outputs the left parenthesis character.

MEM **(MEM)**

Prints out the number of free object entries, the number of free List entries and the amount of unallocated memory (in bytes). Returns the amount of unallocated memory.

MULTIPLY **(MULTIPLY n1 ... nn)**

Returns the product of *n1* times *n2* ... times *nn*. Note that no check is made for overflow.

NEW **(NEW)**

Clears memory of any user defined Atoms, Lists, and functions without changing settings such as pretty print, echo, or number base.

NIL **NIL, (NIL), or ()**

This is the Lisp specification of 'nothing'. As a value it returns *nil*; as a function it also returns *nil*. It is also considered both an Atom and a List.

NTH **(NTH n l)**

Returns the *n*th element of the List *l*.

NULL **(NULL x)**

Returns *t* if *x* evaluates to *nil*, returns *nil* otherwise (performs a logical NOT function).

NUMBERP **(NUMBERP x)**

Returns *t* if *x* evaluates to a number, *nil* otherwise.

OR **(OR x1 ... xn)**

Evaluates *x1* and if it is *nil*, *or* proceeds to evaluate the following arguments until a non-*nil* value is returned, or the end of the argument list is encountered. Note that *or* does not evaluate the following arguments if non-*nil* is returned from any argument.

PATOM **(PATOM a)**

Prints the Atom *a*.

PRINT **(PRINT x)**

Prints the expression *x* evaluates to, followed by a carriage return. *Print* returns the value *t*. If the pretty print flag is set, each parenthesis level will be started on a new line, and will be indented.

PROGN **(PROGN (x1) ... (xn))**

Successively evaluates the specified function expressions, *x1* through *xn*.

PUTPROP **(PUTPROP a pn pv)**

Puts the property value *pv* as the property *pn* of Atom *a*. If the property *pn* already exists, the old value is replaced by the new value.

QUOTE **(QUOTE x)**

Returns the unevaluated expression *x*.

RATOM **(RATOM)**

Waits for a single Atom to be entered from the terminal.

READ **(READ)**

Reads an expression from the terminal.

RPAREN **(RPAREN)**

Outputs a right parenthesis to the terminal.

SAVE **(SAVE fn)**

Saves the current environment (including all objects and Lists) to file *fn*.

SET **(SET a x)**

Causes the value of *x* to be assigned to the Atom *a*.

SETBASE **(SETBASE n)**

Sets the number base used for conversion for both input and output. Note that *n* is converted and evaluated with the *current* base before the new number base is set.

SETDEBUG **(SETDEBUG x)**

If *x* evaluates to *nil*, debug mode is turned off; if it evaluates to non-*nil*, debug mode is turned on. Debug mode is useful for problem determination. If debug mode is on, Micro-Lisp will track up to the last 128 functions. This tracking can be reviewed using the *baktrack* function. Unlike *trace*, which displays the actual input arguments to a function, *debug* only tracks the unevaluated arguments. Debug mode also gives you an opportunity to print any Atom values before *lambda* and function arguments are restored in error processing. This is sometimes useful in determining what caused an error condition to occur.

SETECHO **(SETECHO x)**

Used to control the echo of *source* input to the terminal. *(Setecho t)* will cause all *source* input to be echoed (this is the default setting). *(Setecho nil)* will eliminate the echo.

SETPRETTY **(SETPRETTY x)**

Used to set the pretty print flag. If *x* is non-*nil*, pretty printing will be turned on and expressions will be printed with each parenthesis on a new line and indented for easier reading. If *x* is *nil*, the flag is turned off.

SETQ **(SETQ a x)**

Causes the value of *x* to be assigned to the Atom *a* (similar to *set*, except that for *setq*, *a* is not evaluated).

SOURCE **(SOURCE fn)**

Directs that input is to come from the disk file *fn* instead of the keyboard. Input is obtained from there until the end of the source file. The source file may contain another *source* statement. This will close the current source file and open the new source file for input.

SUBTRACT **(SUBTRACT n1 n2)**

Returns the value *n1* - *n2*.

SYS **(SYS adr x y a f)**

Invokes a machine language routine at address *adr*. The values for the *x*, *y*, *a* and *f* (flag) registers are optional and specified by the corresponding arguments. This function will not allow the IRQ flag to be set, but any other processor flags can be set with the *f* argument. This function returns a List of numbers consisting of the values of the X, Y, A and FLAG registers after the return from the machine code call.

T **T, (T)**

T is one method of specifying a non-*nil* value. As a function, *t* returns *t*.

TERPRI **(TERPRI)**

Causes a carriage return to be output.

TRACE **(TRACE x)**

Causes each function level and its arguments to be output to the terminal as it is evaluated. Note that the actual evaluated arguments are output.

UNDEF **(UNDEF a)**

Removes the function definition from the Atom *a*. If *a* was a native function that had been redefined, the native definition will be restored.

ZEROP **(ZEROP n)**

Returns *t* if *n* is equal to zero; returns *nil* if it is not.

+ **(+ n1 ... nn)**

Shorthand for *add*.

- **(- n1 n2)**

Shorthand for *subtract*.

&#42; **(* n1 ... nn)**

Shorthand for *multiply*.

(/ n1 n2)

Shorthand for *divide*.

Special Input Characters

^ Used to start and end a comment. All characters between the first ^ and the second ^ are ignored. (Note: this character prints on your C64 screen as an up-arrow.)

' Used as a shorthand for (*quote* ...) - (*quote x*) can be shortened to *'x*. Note the dropping of the word *quote* and a set of parentheses. *'X* will print out as *(quote x)*.

" Used to allow special characters and spaces to be included in an Atom name. Any characters between the double-quotes will become part of the Atom name. The double-quotes themselves will not become part of the name.

$ Used to specify a base 16 (hex) number regardless of the current setting of *base*. Must precede any digits.

. Used to specify a base 10 (decimal) number regardless of the current setting of *base*. Must precede any digits.

% Used to specify a base 8 (octal) number regardless of the current setting of *base*. Must precede any digits.

Additional notes

All numbers are stored as 24-bit signed integers. This allows a range of +8,388,607 to -8,388,608.

Cursor control keys work as they do in BASIC.