Name: lisp

The lisp command invokes the Multics LISP subsystem. This is a Multics-compatible implementation of the LISP language, in an interpretive environment for on-line use.

LISP is a simple recursive language, well adapted to problems requiring manipulation of structured data, where the structure of the data is more important than the data itself. The Multics implementation is designed to be fast, yet not limited by storage capacity as many other lisp systems are.

Usage

lisp -option-

1) option — is a pathname for a user-provided "saved environment". If not specified, the standard LISP initial environment will be provided. The user may save an environment for this purpose from inside the lisp subsystem. This environment will be used to initiate the user's predefined atoms, etc.

Notes

In the following sections, the conventions and differences of implementation which set Multics LISP apart from other LISP implementations will be described. The reader is referred to any good LISP manual for further information.

An interim version of a Multics Lisp Programmer's Manual is available. The user desiring specific description of the LISP functions as implemented on Multics would be well recommended to refer to this document.

Using the Multics Lisp Subsystem

Executing the lisp command initializes the user's environment, loading the predefined functions and initializing the evaluator/supervisor. The subsystem is designed to be interactive, and upon entering the subsystem, the user finds himself typing to the supervisor, which is just a direct interface to eval. Some typical console output follows:

```
   (user input will be preceded by an arrow - "-->")
-->lisp
   Multics Lisp Version 1
   75 predefined atoms.
```

```
-->(cons (quote a) (quote b))
   (a . b)
-->(plus 1 2 3)
   6
-->(print (quote foo))
   foo
   foo
-->(setq foo 5)
   5
-->foo
   5
-->(putf (quote factorial) (quote (lambda (x) (cond
-->((zerop x) 1) (t (times x (factorial (plus x -1))))))))
   (lambda (x) (cond ((zerop x) 1) (t (times x (factorial
   (plus x -1))))))
-->(factorial 5)
   120
-->(quit)
   r 1850 4.79 28+389
```

The above example points out several things not mentioned before. First of all, the current implementation has an eval-type supervisor, as opposed to the CTSS LISP which had an evalquote supervisor. This supervisor reads 1 s-expression from the current default input stream, evaluates it as a form, and prints the resulting value. It should be noted that all arguments to the function at the top level of the form are evaluated before applying the function -- hence the explicit quoting in the example above.

As in several other large Multics subsystems, LISP has a program_interrupt handler, which allows the user to quit out of a poorly running program, and resume at the top level by typing "pi" as the first command after a quit. This will restore the bindings of the user's variables to their zero-level values.

Input Format for S-Expressions

Input format for an s-expression is much the same as in other LISP implementations, except that commas are treated just as other alphabetic characters, thus forcing atoms to be separated by spaces. The format for integers is an optional sign, followed by a string of digits. Floating point numbers are as in pl1, except that at least one digit must appear on each side of the decimal point. Character strings are formated as in pl1 also, inside double quotes, with the escape of a pair of double quotes for a double quote within the string.

There is an escape convention which allows the creation of atoms with names containing special characters. Any character preceded by a "\" will be treated as an ordinary alphabetic

character, and the "^" will be ignored, unless the "^" character appears in a quoted string.

The user will also note that as a matter of convenience, excess right parentheses will be ignored by the read functions on input. This eases the burden of parenthesis counting placed on the user by the fact that Multics can only read whole lines at a time from the teletype. In addition, lisp maintains an internal buffer, so two s-expressions to be evaluated may appear on the same line. A newline in an s-expression will be treated as a space, as will a tab, unless explicit escaping is done(see the section on input/output).

Leaving the Lisp Subsystem

As seen in the above console example, the function "quit" with no arguments destroys the user's current environment, and causes the user to return to Multics command level.

The Lisp Language as Implemented on Multics

The Lisp language as implemented on Multics is intended to be an upward compatible extension of LISP 1.5 as described in The LISP 1.5 Programmer's Manual by McCarthy et al. There are several areas of essential difference, which shall be discussed in the current section:

1. Character Strings have been added to the types of objects which may be referred to, and subrs to deal with them have been defined.

2. Normal atoms (atomic symbols, not numbers, character strings, or nil) contain five value cells, instead of the old property list. (It turns out that one of these cells is called a property list, but the printname, functional value and apval will not appear on this list)

3. Because there is one value cell for functional value, and because there seemed to be a need for more general specification of properties like FEXPR, EXPR,FSUBR, and SUBR, a modified concept of function definition has been introduced. In most cases the new concept will not differ appreciably, except perhaps as a notational difference.

Character Strings as Objects

Multics Lisp defines character strings as a special data type, just as in most other Lisp systems, arithmetic values are defined as special data types. This extension allows the user to write programs which manipulate character strings much as he would in PL/I or some other procedure oriented language. In other systems, much the same effect is obtained by special functions which operate on the printnames of atomic symbols (i.e., pack, unpack, etc.); but these functions have the unwanted side effect of creating what are called Truly Worthless Atoms on the MAC/AI Lisp system (note that a truly worthless atom is one which is not referenced by any list structure which is currently active, except the oblist, and which has no property list or values).

With these considerations in mind, the character string data type was defined, with its own set of functions. With respect to most operations, character strings behave much as numbers; they evaluate to themselves, and while they do not in general share the same storage as other identical character strings, the function eq compares character strings by value rather than by name(address).

Value Cells and the Pushdown List

As mentioned above, Multics Lisp defines an atomic symbol as a 5-tuple consisting of 4 value cells and a printname. The values contained in the value cells are as follows:

1. The current value binding of the atomic symbol (variable).

2. A "property list". This can be used in any fashion the user desires.

3. The current global value(APVAL). It is not clear that this value, defined in LISP 1.5, is of any practical use. However, there is no suitable way to replace its function with respect to the evaluation of atoms, so it is included for compatibility.

4. The current functional value of the atomic symbol. This cell replaces the properties FSUBR, SUBR, FEXPR, and EXPR of LISP 1.5.

A set of primitive functions is defined, members of which manipulate these value cells. These functions are described in a later section.

With respect to the evaluation of free variables in functions passed as arguments to interpreted expressions, and functions returned as values of other functions: the user is

warned that the current Multics Lisp implementation does not solve the so-called funarg problem, as value bindings are pushed onto and popped from a stack as functions are entered and exited, and a function does not carry the environment in which it was created around with it.

Functional Values

There are several types of objects which may appear as the functional value of an atomic symbol. We may classify them by several properties. The first property is whether or not the function is a compiled subr or an s-expression to be interpreted. This is determined by the type of the functional value, which may be either a special data type called a subr value (that is in fact a type of program link to be snapped with the appropriate compiled code), or an atom or an s-expression to be interpreted.

The second property is the way the function desires to recieve its arguments. It essentially has the option to take a certain fixed number of arguments (like an EXPR or SUBR in LISP 1.5) or a list of all of the arguments presented to it In the case of interpreted functions, there are a few more options, depending on the format of the formal parameter list in the lambda or nlambda expression defining a function. More about that later.

The third property is whether or not the function wants the arguments passed to it to be evaluated. The user of other Lisp systems should note that this is a consideration independent of the second property mentioned above. A subr value contains information as to whether or not it expects its arguments evaluated. In the case of interpreted functions, a lambda expression always recieves its arguments evaluated, and an nlambda expression always recieves its arguments unevaluated. Certain functions which are handled as special cases in the evaluator may violate this rule, and evaluate some but not all of their arguments (e.g. setq).

Lambda and Nlambda Expressions

The concept of lambda and nlambda expressions is a generalization of the EXPR-FEXPR property found in most other lisps, and the Multics version attempts to be a generalization of the limited, but similar, concepts of the same name in D. Bobrow's two documented Lisp implementations. In terms of the user, perhaps the facility is only a notational change, but the idea has some generality of its own.

A lambda expression has the following syntax: a list of the atom lambda followed by a formal parameter list followed by a

series of n >= 1 forms to be evlaluated, the last of which will generate the value returned by the lambda expression when applied.

The formal parameter list may be the atom nil, in which case the function expects no arguments, or else any atomic symbol, in which case the atomic symbol is bound to a list of the evaluated arguments, or else a list of one of two flavors. If the list is of the form (a b c d) where a, b, c, and d are atomic symbols, the function expects exactly four arguments, and a, b, c, and d are bound to the values of their corresponding arguments. This is identical to the LISP 1.5 definition. However, the formal parameter list may be of the form (a b c . d), in which case the function expects at least 3 arguments, and a, b, and c are bound to the values of the first 3 arguments respectively, but d is bound to the list of the rest of the evaluated arguments, or nil if there are only 3 arguments.

Nlambda expressions are exactly similar to lambda expressions, except the atom nlambda appears instead of lambda, and arguments to the function are not to be evaluated.

Examples

The following may help the user understand the above discussion. The function list which evaluates its arguments and returns a list of them may be defined as follows:

```
(lambda list_of_args list_of_args)
```

The function prog1 which evaluates all of its arguments and returns the first may be defined as follows:

```
(lambda (arg1 . rest_of_args) arg1)
```