I. Introduction

The Multics Lisp subsystem is intended to provide the user of Multics with access to an interpretive Lisp subsystem on that operating system, and to provide the user of Lisp with a Lisp system which has the ability to use the power of the Multics operating system to its own advantage. Certain features stand out in the design of this Lisp implementation, but none so much as the ability to create arbitrarily large list structures without the fear of running out of room which is present on other systems in which lisp is implemented.

The efficiency of a value cell representation of atoms is added to a unified storage scheme in order to hopefully provide the user with an inexpensive, powerful language interpreter.

The current version is somewhat incomplete, and some functions included in this document have not been added to the environment yet. The sections of text which refer to unimplemented features will be surrounded by square brackets ([]). These will be removed as the system is improved.

II. Using the Multics Lisp Subsystem

The Multics Lisp Subsystem is available to all users through the "lisp" command. This 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
   lisp
   70 atoms defined.
-->(cons (quote a) (quote b))
   (a . b)
-->(plus 1 2 3)
   6
-->(print (quote foo))
   foo
   foo
-->(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.

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).

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.

III. 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 se t 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)
```

IV. Primitive Functions and Predicates

The following is a list of the primitive functions defined in the Multics Lisp initial environment. Other functions defined in the initial environment will be found in succeeding chapters.

car,cdr,caar,cdar,caar,cddr - the standard definitions still hold for these, but in the case of nil, all return nil.

cons - forms a dotted pair of its two arguments. cons(x,y) when y is a list, appends x to the beginning of y.

rplaca,rplacd - these functions of two arguments actually modify the internal list structure while performing an operation essentially equivalent to (lambda (x y) (cons x (cdr y))) in the case of rplaca, and (lambda (x y) (cons (car x) y))) in the case of rplacd.

quote,function - these functions of one argument return their unevaluated argument. The primary use for them is shielding their argument from evaluation before being passed to another function. function is equivalent to quote at the present time, and exists only for compatibility. Once again the user is warned that the current implementation does not solve the funarg problem.

cond - cond takes any number of arguments, which should be lists themselves, and selects the first list whose car evaluates to a non-null value, then evaluates the cadr, caddr, etc. of this list as forms and returns the value of the last form evaluated. If the cond "falls through",i.e. the car of all the arguments to cond evaluates to nil, then cond returns nil.

progn - this subr takes any number of arguments, evaluates all of them in order, and returns the value of the last argument evaluated. This function is a generalization of the function prog2 found on most lisp systems.

prog - this function implements the program feature of lisp. It takes at least 2 arguments, the first of which is a list of program variables which will be rebound to nil at the entry to the prog. The rest of the arguments form a sequence of "statements" and labels which will be evaluated in order. A "statement" is a non-atomic item, which will be evaluated in proper sequence at the time the prog is executed. A label is an atomic symbol, which labels the next statement in the list, and which may be used as the target of a goto via the go function.

go - this function implements the goto in the lisp program feature. Its argument must be an atomic symbol whic it will not evaluate, but which designates the point to which transfer of control is to be made. Currently, no non-local goto's may be executed. This includes goto's attempted in a higher invocation of eval or errorset than the one in which the label was defined.

return - this function causes a return from the prog which was most recently entered. It takes 1 argument, which it evaluates, and returns as the value of the prog.

set - this function causes the value of its first argument to be set to the value of its second argument. Note that the first argument must evaluate to an atomic symbol, else this is an error.

setq - like set, but its first argument should be an atom, which is not evaluated, but instead the value of the second argument is assigned to be the value of the atomic first argument.

setqq - like setq, but its second argument is not evaluated.

eval - this function takes its single argument evaluated and evaluates that, returning the resulting value.

cset - primitive function to set the apval value cell. It evaluates both arguments, and places the value of the second arg in the apval cell of the atomic symbol which is the value of the first arg..

putf - primitive function to set the functional value cell. It evaluates its two arguments, the first of which must evaluate to an atom, and then places the value of the second argument in the functional value cell of the first.

Predicates

eq - predicate which tests for identity between its two args, which are evaluated. Identity is equality of value for numbers and character strings, but for atomic symbols and lists, identity holds for the same object only. That is, two lists which print the same may not test identical.

null - evaluates its argument and returns t if argument = nil and nil if argument not = to **nil**.

atom - evaluates its argument and returns t if argument value is a number, character string, or atomic symbol. Otherwise, it returns **nil**.

numberp - evaluates its argument and returns t if its argument has a numeric value.

fixp - like numberp, except returns t only for fixed point (integer) values.

floatp - like **fixp**, but returns t only for floating point values.

stringp - like **fixp**, but returns t only for string values.

VII. Input/Output Functions

The input/output functions of Multics Lisp deal with Multics standard I/O streams. This allows the user to specify the disposition of his output, or the source of his input, by the current attachments of stream names known to the IO switch. Currently, input and output may be performed by Lisp on any character oriented device, through any outer module the user might specify in an attachment. In particular, the user console and ascii segments may be sources (or sinks) for Lisp input (output).

The set of functions which deal with input/output generally fall into two classes: functions which read (or write) and convert the items read (written) into an internal representation suitable for manipulation by other Lisp functions, and functions which control the operation of input/output in general (i.e., perform attachments, detachments, format control, etc.).

One other concept needs to be mentioned. The input and output functions of Multics Lisp either specify a stream name explicitly (as in read_stream), or implicitly (as in read). The implicit specification of a stream is assumed to refer to one of two default streams. These streams are initialized to the streams "user_input" and "user_output" upon entering lisp from Multics command level. The user may change the setting of the default streams from inside Lisp, however. This proves most useful when the user desires to load a set of function

definitions from an ascii file in his directory, since the input to the evaluator/supervisor is taken from the default stream. The user should also note that a stream name is specified by a Lisp character string value, or by the atom nil (which specifies that the default stream is to be used).

Functions

read - this function reads one s-expression from the currently defined default input stream. It takes no arguments.

read_stream - this function works like read, except that it takes one argument which must evaluate to a character string; this specifies the stream to which the function is to read.

print - this function takes one argument which is evaluated, then printed on the current default output stream, followed by a newline.

print_stream - this function works like print , except that it takes an additional argument which must evaluate to a character string; this will be the stream to which the output is directed.

ratom - reads one atom from the current default input stream. Right and left parentheses and periods are read as individual atoms. The function takes no arguments and returns the atom read.

ratom_stream - is to ratom as read_stream is to read.

prin1 - this function works exactly as print, except no newline follows the output.

prin1_stream - is to prin1 as print_stream is to print.

Input/Output Control Functions

These functions control the operation of Multics Lisp I/O. Since Multics Lisp I/O is kept as identical as possible with Multics I/O, these functions can affect more than just the I/O performed by the user of Lisp. For instance, it is possible for the user to close his teletype output, which essentially detaches the user's teletype from his current Multics process. This being a dangerous thing to do, the user is cautioned to use the control functions described below with great care.

output - this function takes an argument which evaluates to a character string or nil. If the argument is a string, the old name of the default output stream is returned, and the new value of the default output stream is that of the argument to output. If the argument is nil, the current value of the default output stream is returned, but no resetting is done.

input - this function works like output, except that it sets the default input stream.

infile - this function opens (attaches through the file DIM) the file specified as the argument to infile, for input. It should be noted that the read,ratom,print, etc. functions will attempt to open the stream specified if it has not already been opened. The value returned is the value of the argument if the attachment was performed successfully, nil if not.

outfile - like infile, except that the file is opened for output.

closefile - this function closes (detaches) the specified ioname. If the argument is nil, the current default output stream is closed, else the one specified by the character string value of the argument is closed. The value returned is either the name of the stream closed, or nil if the stream could not be closed.

openp - is a predicate which takes a character string argument or nil. If furnished with a character string, it returns true if that string corresponds to an open I/O stream. If the argument is nil, it returns true if the current default output stream is open.

printlevel - function which takes one argument, either a number or the atom nil. It sets the current printlevel to the value of the argument and returns the old printlevel value as an integer. If the argument was nil, the value of the printlevel is not changed, but its current value is returned.

VIII. Arithmetic Functions

The arithmetic functions defined on Multics Lisp take both integer and floating point arguments. Most of them are defined in such a way that if all arguments are integer, then the result is an integer, unless the result grows to be > 2 ** 17 - 1, in which case the result is converted to floating point. If not all arguments are integers, then the result of a function is float. In all cases the arithmetic functions evaluate their arguments.

Functions

add1 - this function adds 1 to its argument.

sub1 - this function subtracts 1 from its argument.

plus - takes a variable number of arguments and adds them all together

difference - takes 2 arguments and subtracts the second from the first

times - takes variable number of arguments and returns the product of then all

quotient - takes 2 arguments and divides the first by the second. If both fixed point, then a fixed point divide is used, else a floating point divide is used.

remainder - takes 2 arguments and divides the first by the second as in quotient. If both fixed, then the number theoretic remainder is returned. Else floating 0.0 is returned.

minus - takes 1 argument and negates its value.

abs - takes 1 argument and returns its absolute value.

Predicates

zerop - returns t if its argument is integer 0 or floating point zero.

greaterp - true if first argument greater than second. Fixed point comare is used if both arguments are fixed, else a floating compare is used.

lessp - like greaterp but true if first argument less than second.

minusp - like zerop but true if its argument is less than 0.

Conversion Functions

fix - takes 1 argument and returns the value if the value is fixed, or the integer part of the value if the value is float.

float - takes 1 argument, and returns the value if the value is float, or the floating point number of equivalent value if the number is fixed.

X. Error Handling and Debugging Functions

Errors are handled by Lisp in the following way: 1) print out an error message, 2) reset all of the bindings of atoms to the way they were at supervisor level, and 3) return to supervisor level.

The user has the option to modify the above procedure by the functions mentioned below. Basically, one may set the level to which the error routine will return. Of course, the bindings of variables will only be reset as far as the level of return.

Errors are normally signalled by the lisp system, or by some routine calling the function error. However, the user may introduce a "strong" error at any time, by hitting the attention button on his console, and then typing "pi", which signals a program interrupt to the lisp system. This is handled by executing the normal error procedure, but ignoring errorsets (thus forcing the user back to supervisor level).

Error_Handling_Functions

errorset - this is the standard function for resetting the error handling procedure of the interpreter. It takes 1 argument, which is evaluated, and evaluates it, after setting the level for error returns. If no error occurs, errorset returns with (cons result nil), but if there is an error, errorset returns nil itself.

ersetq - this function is like errorset, except that it does not recieve its argument evaluated.