INFORMATION INTERNATIONAL INCORPORATED
NEW YORK OFFICE

LISP II Project
Memo 8

April 13, 1965

INPUT-OUTPUT FOR LISP II

by
Paul Abrahams

1. INTRODUCTION

This memo is intended as an augmentation and modification of the input-output system proposed by Clark Weissman in an earlier memo. By and large I have followed a similar organizational scheme, which should make comparisons easy. I have also used Clark's functions and conventions wherever possible.

There are three criteria that the input-output scheme presented here has been designed to fulfill:

(a) It must be possible to convert an input program rapidly into a stream of tokens that form the input of a syntax translator.

(b) It must be possible to read and write formatted data a la FORTRAN. The FORTRAN capabilities in this area are a bare minimum; NPL is a far better comparison.

(c) There must be a set of primitive operations that enable the programmer to position input-output devices and main memory without performing any interpretation of the meaning of the data being read or written, or of the meaning of record gaps, file marks, etc.

The first area covered by this proposal is a set of input-output primitives for the selection, deselection, positioning, reading and writing of input-output devices and for the allocation of buffer space for these devices. The second area is the definition of the finite state machine that is used to convert character streams into tokens. Unlike the earlier proposal, the finite state machine makes explicit use of format information, and is designed to handle both the reading of programs to be translated and the reading of data called for by the program. The finite state machine takes as its input an atom of type string together with a procedure for obtaining a new string when the old one is exhausted. There is thus no necessary logical connection between the input-output primitives per se and the finite state machine, even though they ordinarily will be used in tandem.

One area that is not treated here is the preparation of

output strings. It may prove desirable, when this problem has been analyzed further, to modify the present finite state machine scheme so as to have greater symmetry between input and output. This possibility will be discussed in connection with the finite state machine.

The primitives proposed here are quite specifically oriented towards the structure if not the details of the input-output system used by the Q32 time-sharing system. Some modifications would be required if it were to be meshed with an input-output environment such as IBM's IOCS or IBSYS; however, both IOCS and IBSYS are strongly oriented towards batch processing rather than interactive computing. The essential difference between TSS and IBSYS in this area is that in IBSYS one can release a file without releasing the physical unit housing that file; in TSS one cannot. However, the major purpose in releasing files but not units in IBSYS is communication between successive segments of a job; and ordinarily that necessity does not arise in a time-shared environment.

2. DEFINITION OF THE ENVIRONMENT

2.1 Files and Buffers

The term file will in general be used here in the same sense that it is used by the Q32 dispatcher, namely, as a unified collection of data stored on an external medium. Some input-output devices, such as discs, can have more than one file on a physical unit; others, such as tapes, cannot. Each file will have a logical name, which corresponds with the name used by the Q32 file call. For names of more than six characters, LISP will automatically substitute a generated name. CTSS (Project MAC) file designations, which consist of a first name and a last name, can be made into a single name by concatenating the two names with a period between them.

A particular file may be either active or inactive. A file is active if it has been requested through a Q32 dispatcher file call and not released. The user will define a file before he activates it and the definition will persist after he releases it; thus he may also have inactive files. An active file may or may not have buffer space assigned to it. Buffers for files will be dynamically relocatable arrays, and will be assigned at run time. Binary files will use arrays of type integer and BCD files will use arrays of type string. An attempt to read from or write onto a device without assigned buffer space will cause an error complaint; however, such a device can be repositioned without causing an error.

2.2 Data Format and Types

Three kinds of delimiters will be recognized by the input-output primitives: record marks, file marks, and data terminators (e.g., end of tape). These delimiters will correspond to physical rather than logical delimitations of the data. The input-output primitives will operate independently of whether the data being handled is binary or BCD, except for the definition of a file. BCD data will be assumed to be packed eight 6-bit characters to a word for the Q32 system and will be modified appropriately for other systems.

2.3 Unit Identifiers

For each type of device that can be read or written by the system, a LISP object will exist. This object will have on its property list the necessary information about devices of the specified type such as record length, direction(s) of data flow, and binary or BCD data type. The object will be known as a unit identifier; it can be used as an identifier in other contexts with the restriction that certain of its properties cannot be deleted or modified. (There may be times when one wishes to change these properties deliberately, e.g., to change the record length definition for reading a tape with nonstandard blocking; and this kind of change is perfectly

permissible.) Among the unit identifiers may be TAPE (for symbolic magnetic tape), BTAPE (for binary magnetic tape), PTAPE (for symbolic paper tape), BPTAPE (for binary paper tape), DISC, SCOPE, TTY, CONSOLE, PHONE (high-speed data link), COMPUTER, DRUM, CARD (reader or punch), PRINTER, and RANDTABLET.

3. INPUT-OUTPUT PRIMITIVES

The primitives presented here have been designed so as to maximize their independence. Thus, operations have been arranged as far as possible so as not to utilize each other or much of the apparatus of LISP. Some LISP apparatus will be needed for searching property lists and similar tasks. The primitives are divided into three groups: the file definition primitive, the file activation primitives, and the file manipulation primitives.

3.1 File Definition Primitive

3.1.1 DECLARE(LN U P) - This function creates a file definition for the file with name LN on a unit of type U with additional parameters P. All relevant properties of the file are stored on the property list associated with the file name. This function is really not a primitive since it probably can be programmed in LISP, but it is presented here nevertheless to make the conception of the system clearer.

3.2 File Activation Primitives

3.2.1 REQUEST(LN) - This primitive requests the activation of the file named LN, and causes a file call to be sent to the dispatcher. All information necessary to construct the file call is contained on the property list of LN. No buffer space is assigned by this primitive.

3.2.2 RELEASE(LN) - The file named LN is deactivated, and a DEFILE call is sent to the dispatcher. Buffer space held by the file is given back to LISP for cannibalization.

3.2.3 BUFFER(LN A) - The array A is assigned as buffer space for the file named LN. If A is not sufficiently large, an error complaint results. The programmer must obtain the array A via LISP.

3.2.4 UNBUFFER(LN) - The buffer space held by file LN is released to LISP.

3.2.5 SAVEDSC(LN LN1) - The disc file named LN is stored permanently on the disc under the name LN1. This and the next two file primitives have been included so that the LISP II user can gain access to the full facilities of the dispatcher. I would suggest that as new dispatcher calls are added, new LISP primitives be created. I propose this reluctantly, but there does not seem to be any satisfactory framework into which all the file calls can be fitted in a general way.

3.2.6 RENAMEDSC(LN LN1) - The permanent disc file LN1 is made into the private file LN, using the dispatcher REFILE

call. LN must have been declared; the RENAMEDSC call automatically activates LN as if it had been REQUESTed.

3.2.7 DELETEDSC(LN1) - The permanent disc file LN1 is deleted from the disc. The name LN1 cannot have more than six characters, or whatever other limitation is imposed by the file call. LN1 will not ordinarily correspond with a file requested by the programmer; in fact, no check will be made as to whether the programmer has activated such a file. The motivation here is that DELETEDSC should be used for scratching library files and RELEASE for scratching private files.

3.3 File Manipulation Primitives

3.3.1 POSITIONR(LN N) - The device assigned to LN is advanced by N physical records if N is positive and backed up by N physical records if N is negative. If a file mark is encountered the device is not moved further. The value of the function is EOF if a file mark is encountered, EOT if end of logical file (i.e., the entire data file) is encountered, BOT if beginning of logical file is encountered, and NIL otherwise.

3.3.2 POSITIONF(LN N) - The device assigned to LN is advanced by N physical files if N is positive and backed up by N physical files if N is negative. The count is advanced by one whenever a file mark is encountered; when the count equals N, the device is positioned just after the file mark. The value of the function is EOT if the end of the logical file is encountered, BOT if the beginning of the logical file is encountered, and NIL otherwise.

3.3.3 MOVEIN(LN) - One physical record is moved from the device assigned to LN into the buffer assigned to LN. An error complaint results if no buffer has been assigned. The value of the function is the same as for POSITIONR, except that REDUND indicates an unreadable file. If no buffer space has been assigned to the file, an error complaint results. An alternative approach would be to have MOVEIN automatically obtain buffer space if the file is not assigned any; however, this alternative would permit buffer-eating programs to survive undetected. The programmer, of course, can always write a function MOVEIN1 that tests to see if LN has buffer space and if not assigns it, so that the error complaint can be circumvented if one wishes.

3.3.4 MOVEOUT(LN) - One physical record is moved from the buffer assigned to LN to the device assigned to LN. An error complaint results if no buffer has been assigned. The value of the function is REDUND if the record cannot be written, and NIL otherwise.

3.3.5 WEOF(LN) - A file mark is written on the device asso-

ciated with LN. LN need not have buffer space.

3.3.6 REWIND(LN) - The device assigned to LN is moved to its initial position.

3.4 Specification of Current File

The programmer, if he wishes, can create own variables INFILE and OUTFILE whose values are the specific files currently being used for input and output respectively. He can also create auxiliary input-output functions that lack the parameter LN and substitute for it the bound variable INFILE or the bound variable OUTFILE. Thus he need not specify the unit involved when he calls for input-output. (It should be noted that he will need some way to indicate whether positioning is to be done on the input file or on the output file.)

3. FINITE STATE MACHINES

The input-output primitives given in the previous section enable the programmer to bring data from an external source into the computer in the form of string arrays, but do not provide him with a mechanism for interpreting these strings. Such a mechanism could be built using LISP string manipulation functions and not using finite state machines at all; in fact, it may be helpful to avoid the finite state machine during some stages of bootstrapping. The finite state machine is intended to make the process of interpreting input strings more efficient, and at the same time to provide a language for specifying the interpretation in a convenient way.

A finite state machine takes a character stream as input and produces a token stream as output. There are three subprocesses required to do this: unpacking of single characters from the input stream, delimitation of tokens, and translation of character strings that stand for tokens into the tokens themselves. In the scheme proposed here, unpacking and delimitation are performed more or less simultaneously, and translation is performed as a subsequent operation. This arrangement is quite efficient since very little saving and restoring of the accumulator is needed.

Finite state machines are intended to apply only to data of type string. Thus they will not ordinarily be used in conjunction with binary files. However, should the need arise one can always redeclare the type of the resulting array using the LISP type conversion functions, and thus apply a finite state machine to binary data (which will, of course, be grouped into character-sized bytes).

4.1 Specification of a Finite State Machine

The system proposed here consists not of a single finite state machine but rather of a finite state machine interpreter. In order to create a specific finite state machine one provides the interpreter with an appropriate collection of tables and procedures. For cases where the interpretation loss is critical, a particular finite state machine can be hand-coded, using the same program structure as in the more general case. However, the interpretation process is sufficiently simple so that not much loss of efficiency occurs.

The finite state machine interpreter is invoked as a function of no arguments; its value is the next token on the input string. In reality, the interpreter has six arguments, which exist in the form of own variables; for a specific character string and finite state machine these variables are set before interpretation is begun,

and need not be changed from one token to the next. This arrangement avoids the need for spending time in communicating arguments. Its disadvantage is that when two finite state machines are operating simultaneously these variables must be saved and restored for each token; however, this case is sufficiently rare so that LISP can live with this particular inefficiency without great discomfort.

The central part of the specification of a finite state machine may be viewed as a set of quintuples of the form

$$(s_i \; q_j \; f_k \; s_m \; a)$$

in which s_i and s_m are states, q_j is a type of input character, f_k is a format symbol, and a is an action. When the machine is about to process a character it is in state s_i. It determines that the character is of type q_j and obtains from a format procedure a state f_k. With each valid triple $(s_i \; q_j \; f_k)$ there is associated a pair $(s_m \; a)$. The machine executes the action a (which may consist simply of storing the input character on a stack), enters the state s_m, and processes the next character.

Associated with each state is a table of character types, in the form of a 64-word table. Any number of states may use the same table, and in fact a particular table may even be used by different finite state machines. A type is associated with each of the 64 possible characters, and this type (a small integer) can be obtained by table lookup.

Now we can state what the <u>own</u> variables of the finite state machine interpreter are:

(1) An array representing the set of quintuples for the machine.

(2) A procedure for obtaining a new string when the present one is exhausted.

(3) A procedure for obtaining a format code.

(4) The location of the current string head.

(5) The current character count within the string.

(6) The number of characters in the string.

The procedure that obtains the format code can be used for counting; in particular, it can be used to handle fixed-length data fields and as a means of bypassing identification fields such as the last eight columns of a card. The format procedure can be given access to the variables of the finite state machine, including the current character, and thus can be used for many different purposes. It can be as elaborate or as simple as the situation requires.

The array of quintuples is in a machine-dependent form, designed specifically to work in conjunction with a particular interpretation program. However, for any given LISP implementation the rules for constructing the array are fixed and explicit.

One of the actions available to a quintuple is ejection of a token. This action temporarily halts the operation of the finite state machine. When operation is resumed the last character examined is read again. The motivation for this convention is that often the first character of the (n + 1)th token is the terminal delimiter of the n'th token, and in very few cases is the terminal delimiter of a token actually part of the token itself. For these few cases it is easy enough to force the reading of an additional character.

4.2 A Specific Finite State Machine Interpreter

In this section we shall present a sketch of the code for the Q32 finite state machine interpreter. The code is admittedly lavish in its use of index registers, and I have omitted much of the housekeeping. However, the purpose of this code is not so much to present a realistic implementation as to illustrate more or less how efficient the interpretation is and what the conventions are.

We shall use index registers as follows:

XR1 - current byte within word
XR2 - word within array
XR3 - state
XR4 - type of current character
XR5 - format code
XR6 - character stack pointer

The possible actions are:

(1) Place the current character in the stack.

(2) Call a procedure and restart the stack.

(3) Call a procedure and leave the stack undisturbed.

(4) Call a procedure and return its value as a token.

The character stack is simply an array with one character per word. In the code given here we have not dealt with the problem of stack overflow, which can occur since the size of the stack must be fixed. If the stack overflows, the program can requisition a larger array; but figuring out just how to do this seemed premature at this stage of the specification.

In order to find the last two elements of a quintuple

from the first three, we execute the instruction

```
LDB         QUINTS,I
```

The cell QUINTS gives the start of the quintuple array, which is used for a three-stage lookup using three levels of indirect addressing. The first level uses the state as an index, the second uses the symbol type as an index, and the third uses the format code as an index. The actual contents of QUINTS are

```
PZE         LOC1,3,I
```

where LOC1 is the first of a block of cells, one for each state. Thus, if XR3 contains i, the state number, the first level of indirectness will lead us to the cell reserved for this state. Its contents are

```
PZE         LOC2,4,I
```

where LOC2 is the first of a block of cells reserved for state i. The j'th cell of this block, which will be referenced by "LOC2,4", will be assigned to the state-symbol type pair (i,j). Its contents in turn will be

```
PZE         LOC3,5
```

where LOC3 is the first of a block of cells reserved for state i paired with symbol j. "LOC3,5" will, of course, reference the cell within the block corresponding to the format code and giving the last two elements of the quintuple. The actual code word will contain the action code in byte 0, the new state in bytes 1, 2, and 3, and the location of the procedure to be invoked, if any, in bytes 5, 6, and 7. Thus the array of quintuples is divided into three parts: the first for states, the second for state-symbol type pairs, and the third for the state-symbol type-format code triples. The third part actually contains the code words.

In addition, we need an array of cells, one for each state, giving the table used to encode characters into character types. Each cell is of the form

```
PZE         TYPLKP,A
```

where TYPLKP is the beginning of the table for this state. We will assume that the location of this array is in cell TYPTAB. The rough code for the finite state machine interpreter, then, is the following:

```
LOOP    LDS,B7,G07  CHPOSN
        LDX,1       $A
        LDA         CHPOSN
        SFA         (3)R
        LDX,2       $A
        BUC         FMTPR,I         get format code
        LDX,5       $A
        XEC         GETBYT,1        get character
        LDX,4       TYPTAB,3,I      get character type
        LDB         QUINTS,I
        STB         TEMP
        LDX,3,L     TEMP
        TST,B0,G07  TEMP
        BUC         A1
        BUC         A2
        BUC         A3
        BUC         A4
A1      STA         STAK,6          store char in stack
        BAX,6,1     CONTIN
A2      LDX,6       (STAKL)R
        BUC         TEMP,I
        BUC         CONTIN
A3      BUC         TEMP,I
        BUC         CONTIN
A4      BUC         TEMP,I
           (return code)
CONTIN  AOR         CHPOSN
        CMA         CHLIM
        BUC         $+3
        BUC         $+2
        BUC         LOOP
        BUC         NEWSTR          get new string
        STZ         CHPOSN
        BUC         LOOP
GETBYT  LDA,B0      STRING,2
        LDA,B1      STRING,2
                . . .
        LDA,B7      STRING,2

CHPOSN                              character count
CHLIM                               no. of char.'s in string
QUINTS                              loc. of quintuple table
TYPTAB                              loc. of pointers to
                                       char. type tables
STAKL                               beginning of stack
```