Mr. [illegible] [illegible]

2. DATA AND ITS TYPES

2.1 Introductory Definitions

In order to clarify the discussion of LISP II data, we shall first explain the terminology to be used. We will regard a datum as an abstract entity whose precise nature is irrelevant to our purposes, and a program as the specification of an algorithm for processing data. Some data are represented explicitly in the program; these are constants. Some data are designated by variables; during the execution of a program, the datum (if any) designated by a variable at a given time is known as the value of the variable (at that time). Some data, such as those designated by subexpressions of algebraic formulae, have no simple designation.

Data may be classified into types, such as "REAL" or "SYMBOL" (for symbolic expressions). A datum has one and only one type. The type of a constant can always be deduced from its form, e.g., "75" is an integer since it consists only of decimal digits. The type of a variable must be declared by the programmer, either explicitly or by omission; the type declared for a variable restricts but does not necessarily determine the types of the value of the variable at different times. In particular, a variable declared to be of type "SYMBOL" may have as values data of different types at different times. LISP differs from ALGOL in this respect.

A datum has an internal representation within the computer as a sequence of bits (or whatever else is appropriate for the computer at hand). The internal representations of LISP data will be discussed in Sec. . The standard (external) form of datum is the external representation that results when the datum is printed, i.e., converted from its internal representation. A datum may have external representations other than its standard form, e.g., "007" is an external representation of the datum whose standard form is "7". Different external representation of the same datum are said to be equivalent.

2.1.1. LISP II Character Set. The LISP II reference language. publication language, and Q32 hardware language are all identical, and thus only one character set need be defined. Were LISP II to be implemented on a different machine, a different hardware language might be necessary.

The LISP II character set contains a total of 60 characters, and is a subset of ASCII. In this manual, the space will be printed as "b̸" and the carriage return (or, more generally, end-of-line) as "ⓡ". The characters may be grouped as follows:

26 letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
10 digits	0123456789
4 bracketers	[ ] ( )
7 operators	+ - * / ↑ ← \
4 punctuators	. , : ;
3 relators	= < >
4 special delimiters	b̸ # $

2 spacers ƀ ®

The five characters

@ & ? " !

are part of ASCII but are not part of the LISP language.

2.2 Available Data Types

The available data types may be grouped into three categories: simple types, array types, and formal types. There are five simple types: "BOOLEAN", "INTEGER", "OCTAL", "REAL", and "SYMBOL". An array type is described by specifying the types of elements in the array. A formal type is specified by specifying the type of its value and of its arguments.

2.2.1 Simple Types.

2.2.1.1 Boolean Data. There are two possible values for a Boolean datum: "TRUE" and "FALSE". "TRUE" and FALSE" are Boolean constants with the obvious meanings. The constants "()" and "NIL" are equivalent to "FALSE"; "FALSE" is the standard form.

2.2.1.2 Integer Data. An integer datum is represented either as a series of decimal digits preceded by an optional sign and followed by an optional exponent, or as an exponent standing by itself. The exponent is written as the letter "E" followed by a series of decimal digits. The exponent indicates multiplication by a power of 10, e.g., "3E2" is equivalent to "300". When an exponent stands by itself, it represents a power of 10, e.g., "E5" is equivalent to "100000". "+0" and "-0" are equivalent. The size of an integer datum is limited by the word size of the computer (2^{48}-1 in the case of the Q32).

Examples: 279

-05831E4

E16

+490

2.2.1.3 Octal Data. An octal datum is a 48-bit quantity. Octal constants are written as an optional sign followed by a series of octal digits followed by the letter Q followed by an optional scale factor. The scale factor specifies the number of trailing octal zeros to be added, e.g., "15Q7" is equivalent to "150000000Q". A plus sign is ignored, and a minus sign is OR'ed into the leftmost bit. The octal data "+0" and "-0" are distinct.

Examples: -77777Q12

0Q

004Q

-00000Q2

+12345670Q1

2.2.1.4 Real Data. A real datum is represented internally as a floating point number. Real constants are written as an optional sign followed by a mantissa followed by an optional exponent. The mantissa consists of a series of decimal digits containing an interspersed decimal point: the decimal point may be at the beginning or at the end as well as in the middle. The exponent is represented as the letter "E" followed by a signed integer, and indicates multiplication (for a positive exponent) or division (for a negative exponent) by a power of 10. The integer in the exponent cannot itself be exponented. A real constant can be distinguished from an integer constant because only a real constant can contain a decimal point. "-0." and "+0." are equivalent.

Examples: 245.9
367.E-014
.00005
-1.
+2.E+2
-0.

2.2.1.5 <u>Symbolic Data</u>. A symbolic datum, i.e., a datum of type "SYMBOL", is the LISP II analogy to the S-expression of LISP 1.5. "SYMBOL" as a data type really means "symbolic", and does not imply that the datum is a single symbol. The relationship between symbolic data and symbolic constants will be discussed in Sec. 2.2.1.5.3.

Symbolic data are built up from <u>atoms</u>, which are of two kinds: inherent and derived. An <u>inherent atom</u> is either an identifier (Sec. 2.2.1.5.1) or a string (Sec. 2.2.1.5.2). A <u>derived atom</u> is a symbolic datum resulting from the type conversion of a nonsymbolic datum. Every nonsymbolic datum has a corresponding symbolic datum. If an operation is defined for a nonsymbolic datum, then it is defined for the corresponding symbolic datum and has the same result. Thus a symbolic datum is essentially isomorphic to the nonsymbolic datum from which it is derived. Furthermore, when a nonsymbolic datum is converted to a symbolic datum its external representation remains the same but its internal representation differs.

"SYMBOL" serves as a universal type in the sense that a datum of any other type may be treated as though it were of type "SYMBOL". When a variable is declared to be of type "SYMBOL", the values assumed by the variable may be of other types. However, if an operation is performed upon a variable declared to be of type "SYMBOL" and the value of the variable is a nonsymbolic datum, then the datum will automatically be converted to a symbolic datum.

Derived atoms are peculiar in that they cannot be written as constants except as part of a larger symbolic datum. In other words, although "7" is an integer constant, there is no way to write a corresponding symbolic constant. In the context "(A 7)", however, "7" represents a symbolic datum.

The fact that derived atoms cannot be written as constants is of no practical disadvantage since a derived atom is interchangeable with the datum from which it was derived in virtually any context. Although derived atoms cannot be written as constants they can be printed. When printed, a derived atom is indistinguishable from the datum from which it is derived.

2.2.1.5.1 Strings. A string is a sequence of characters. Any member of the character set of the machine may be an element of the sequence. A distinction must be made between a string and the sequence of characters that represents it externally. The representation of a string uses the character "#" as a delimiter and the character "'" as a quoter.

Specifically, the representation of a string begins and ends with the character "#". In obtaining an actual string from its representation, any occurence of the character "'" within the string indicates that the next character is to be taken literally and the "'" itself is to be deleted. The characters "'", "#", and "%" must be preceded by "'" if they are part of the string.

Two particular cases of interest are "'#" and "''", which are transformed respectively into "#" and "'". Thus, the string datum "ISN'T ##5 $" could be represented externally as "#ISN''T '#'#5 $#" or possibly by "#IS'N''T '#'#5 $#"; in the latter example the first prime used is redundant but permissible. Any occurence of "@" within a string is ignored unless the preceding character is a "'", in which case the "@" is included within the string. Thus a string can be spread over several lines.

Examples: #''''''#ƀH AS ƀTHREEƀCHARACTERS#

#7#

#A$B#

#A$®B# (equivalent to #A$B#)

2.2.1.5.2 Identifiers. An identifier is a sequence of characters consisting either of the character "%" followed by a string as defined in the previous section, or of a letter followed by a sequence of characters each of which is either a letter, a digit, or ".". The identifiers "ABC" and "%#ABC#" are identical. Identifiers differ from strings in that they can be used to name functions and variables, and can have property lists associated with them. On the other hand, strings take up less storage space than identifiers.

"TRUE", "FALSE", and "NIL" are not identifiers. "%#FALSE#" is an identifier but is not a Boolean constant.

Examples: YES.WE.HAVE.NO.BANANAS

A257

T98

%#ZOUNDS FORSOOTH#

%#)#

2.2.1.5.3 Symbolic Constants. In order to distinguish the variable "A" from the symbolic datum "A", a quoting mechanism is needed. A symbolic datum is quoted by preceding it with the character "'". Any S-expression in the sense of LISP 1.5

may be quoted. Derived atoms and strings need not but may be quoted, since they can be interpreted without ambiguity. However, identifiers and nonatomic S-expressions must be quoted.

The quoted indentifier A could be written either as "'A" or as "'%#A#". In the case of the single-character identifier corresponding to ")", only the second alternative could be used. Since single-character identifiers are useful in character manipulation, an additional mechanism has been provided for writing them, but only within L_I. Within L_I, a one-character identifier may be written simply by preceding it with a "'". Any single character can be treated in this way. Since the "'" also introduces a symbolic datum, i.e., an expression in L_I, "'')" is a perfectly permissible way to write "'%#)#", and of course a more conveient way to write it.

<u>Examples:</u>
```
''A
#STRING CONSTANT#
'((A . 1) ('$ . 2))
'IDENT
#IDENT#
```

2.2.2 <u>Array Types</u>. An n-dimensional array datum is a set of data such that any element of the set can be singled out by giving <u>n</u> subscripts. Each subscript is an integer whose permissible range is defined by the dimensions of the array. Thus in a 3 x 12 x 2 array the first subscript ranges between 1 and 3, the second between 1 and 12, and the third between 1 and 2.

At present, the lower bound of a subscript of a LISP II array must be 1, though this constraint will probably be removed in the future. When the subscripts specifying an array element are given, the element can be retrieved using efficient table lookup operations; though in principle lists could always be used in place of arrays, arrays are far more efficient both in retrieval time for specified elements and in space requirements.

The elements of an array must all be of the same type. The type of an array is specified by specifying the type of its elements by means of an f-type. An f-type is either the name of a simple type or "FORMAL". "FORMAL" is not a true type, but rather an abbreviation for the family of formal types. Thus the permissible array types are:

BOOLEAN ARRAY
INTEGER ARRAY
OCTAL ARRAY
REAL ARRAY
SYMBOL ARRAY
FORMAL ARRAY

If the elements of an array are themselves arrays, then the array must be declared as "SYMBOL ARRAY". An array constant is written as a left bracket followed by a data type name followed by the elements of the array followed by a right bracket. For a 1-dimensional array the elements are merely enumerated, e.g., "[REAL 3.5 72.9]", which is a single datum of type "REAL ARRAY". The dimension of the array, which is 2, can be deduced from its form. If any of the explicitly given elements of an array are not of the same type as the array itself, they are assumed to be converted to the type of the array itself according to the conversion rules given in Sec. 2.3. Thus the arrays "[INTEGER 4,7]" and "[INTEGER 4.8,7.1]" are identical.

When an array is of type "SYMBOL ARRAY", then its elements are already assumed to be symbolic constants and should not be quoted by using the "'". In the context of a symbolic array, the "'" is used to indicate an array element that is itself an array and thus is given a different interpretation than in any other context.

The procedure for ordering the elements of an array can best be illustrated by the following examples:

```
1-dimensional: [INTEGER 2,5,7,9,3]
2-dimensional: [INTEGER [2,7],[4,9], [0,3],[5,9]]
3-dimensional: [INTEGER
                [[1,5],[    ],[4,7]],
                [[0,9],[1,2],[3,3]],
                [[1,1],[2,7],[9,4]],
                [[8,8],[8,4],[4,0]]]
```

The second example is a 2 x 4 array and the third example is a 2 x 3 x 4 array. The first subscript varies most rapidly and the last subscript least rapidly.

The procedure for enumerating the elements of an array can be stated more formally. We define a subarray of order k to be the set of elements obtained when the last k subscripts are allowed to vary over their entire range and the remainder are held fixed. Thus a subarray of order 0 is a single element; a subarray of order _n_ of an n-dimensional array is the entire array. In listing an array of dimension _n_ we list the successive subarrays of order n-1, separating them by commas and enclosing each one in brackets. Within each subarray of order n-1 we list the successive subarrays of order n-2, again separating them by commas and enclosing them in brackets. We continue in this way until we reach the subarrays of order 0, which are not to be enclosed in brackets.

2.2.6 _Formal data_. A formal datum is a LISP II function. A formal constant is written either in the form of a function declaration, in the form of a lambda-expression, or in the form "FUNCTION _name_", where _name_ is an identifier whose value is a formal constant. We shall not attempt a detailed explanation at this point because an understanding of formal constants requires an understanding of function definitions. Function definitions will be discussed in Sec.   , and formal constants will be discussed in full in Sec.   . For the moment, we merely point out that a formal constant always begins with either "LAMBDA", "FUNCTION", or a data type name followed by "FUNCTION".

2.3 Type Conversions

A datum can be converted from one type to another explicitly or implicitly. Explicit conversion results from the explicit use of a conversion function; implicit conversion results frpm the appearance of a datum of one type in a context where a datum of a different type is expected. Conversion can take place only if certain compatibility requirements are satisfied. These requirements are given in Table 1, which also specifies the nature of the conversion.

Implicit conversions occur under the following specific circumstances:

(1) A datum is assigned as the value of a variable of a different type.

(2) A datum is used as an argument of a function when the function expects a datum of a different type.

(3) A datum is combined with a datum of a different type by an operator that can accept data of more than one type.

In the case of (1) and (2), the datum to be assigned is converted to the type of the variable to which it is assigned. Thus, if a datum of type "REAL" is assigned to a variable of type "SYMBOL", a "REAL" to "SYMBOL" conversion is generated by the compiler. In the case of (3), all data are converted to the most preferred type; the order of preference is, from least preferred to most preferred:

TYPE FROM \ TO	B	I	O	R	S	a-t	f-t
BOOLEAN	X	-	-	-	S	-	-
INTEGER	TRUE	X	IO	IR	S	-	-
OCTAL	TRUE	OI	X	OR	S	-	-
REAL	TRUE	RI	RO	X	S	-	-
SYMBOL	P	SI	SO	SR	X	SA	SF
Array-type	TRUE	-	-	-	X	A	-
Formal-type	TRUE	-	-	-	S	-	F

Remarks:

X	=	exact, no conversion needed
-	=	not permitted
S	=	symbol of appropriate type transmitted
TRUE	=	all non-Boolean values are TRUE
P	=	predicate evaluation: ( ) → FALSE, else TRUE
A	=	array-types must agree, else illegal
F	=	formal-types must agree, else illegal
IO	=	integer-to-octal conversion, exact, except -Ø → +Ø
IR	=	integer-to-real conversion, done by floating the integer
OI	=	octal-to-integer conversion, exact
OR	=	octal-to-real conversion, done by floating the equivalent integer
RI	=	real-to-integer conversion, rounded
RO	=	real-to-octal conversion, rounded
SI	=	if symbol is a number, convert to integer, else illegal
SO	=	if symbol is a number, convert to octal, else illegal
SR	=	if symbol is a number, convert to real, else illegal
SA	=	if symbol is an array and array types agree, transmit the value, else illegal
SF	=	if symbol is a formal-type and formal types agree, transmit the formal, else illegal

OCTAL

INTEGER

REAL

SYMBOL-O

SYMBOL-I

SYMBOL-R

Here "SYMBOL-O" indicates a symbolic datum derived from an octal datum, and similarly for "SYMBOL-I" and "SYMBOL-R". If any of the data being combined are non-arithmetic, then all data are converted to type "SYMBOL".

2.4 Formal Syntax of Constants

The syntax equations for constants, i.e., the rules for writing them down, are as follows:

```
letter        = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
octal-digit   = Ø|1|2|3|4|5|6|7
digit         = octal-digit|8|9
string-character = letter|digit|(|)|[|]|+|-|*|/|\|↑|←|,|.|:|<|>|=|$|V
   (@, ',#,$, and ; are not string-characters.)
character     = string-character |'|#|%|;
Boolean       = TRUE|false
false         = FALSE|NIL|()
number        = integer| real
integer       = octal | decimal
octal         = sign octal-digit {octal-digit}* Q {unsigned-decimal | empty
sign          = +|-|empty
```

```
unsigned decimal = decimal-digit {decimal-digit}*
decimal        = sign unsigned decimal {empty|E|E unsigned-decimal}
real-object = unsigned-decimal .|.unsigned-decimal|unsigned-decimal
                  .unsigned-decimal
real           = sign real-object {empty|E scale}
scale          = sign unsigned-decimal
string         = #{ string-character|;|''|'#|'%|'@} *#
identifier     = letter { letter|digit|.} *|% string
    TRUE, FALSE and NIL are not identifiers.
unquoted constant = number|Boolean|array|string|function-specifier
atom           = unquoted-constant|'character
S-expression= atom
                 |(S-expression {S-expression}* ␢ . ␢ S-expression )
                 |({S-expression}*)
constant       = ' S-expression|unquoted-constant
```

In these syntax equations there are two undefined terms: "array" and "function-specifier". "array" cannot easily be defined in the BNF notation we have used here, but the explanation in Sec. 2.2.2 is sufficient to specify the conventions for writing arrays. "function-specifier" will be defined in Sec.