

TR-72-7
REVISION A
AUGUST 1, 1972



A LIST PROCESSING SYSTEM FOR THE B 6700

By

D.A. Fisher
J.F. Leathrum
M. Luskin

Grace

N.C.E. - U.F. ; J. - BIBLIOTECA		
REGISTRO 909	DATA 7/7/77	FE ANCIAMENTO
CLASSIFICACAO 681.304.28		FORNECEDOR
NF/FAT.	DATA	PRECO <i>Dreco</i>
OBS. <i>Felicio</i>		

Burroughs Corporation
Defense, Space and Special Systems Group

016.F94

LIS

Ex. 2

ENGINEERING
BIBLIOTECA

CONTENTS

Section	Page
1 INTRODUCTION	v
2 DATA STRUCTURE OF THE LANGUAGE	1-1
S-Expressions	1-1
Atoms	1-2
Composite Data	1-3
Lists	1-3
Constructs	1-4
Lambda Expressions	1-4
Concrete Syntax	1-4
2 THE METALANGUAGE	2-1
Abstract Syntax	2-1
Interpretation	2-2
Concrete Syntax and Description	2-2
Primitive Expressions	2-3
Functional Form Expressions	2-4
Infix Expressions	2-5
Reserved Word Expressions	2-6
Programs	2-7
3 THE OPERATION SET	3-1
Definition of Recursive Functions and Syntax Extensions	3-1
Elementary List Processing Operations	3-2
Construct Processing and Assignment	3-5
Environmental, Scope and Definition Operations	3-7
Iterative and Conditional Control	3-9
Arithmetic, Relational, and Boolean Operations	3-11
The Universal LISP Function	3-13
Input-Output Operations	3-14
Conversational and Debugging Facilities	3-16

CONTENTS

Section

Page

4 MEMORY ALLOCATION AND GARBAGE COLLECTION

4-1

5 HOW TO USE THE SYSTEM

5-1

APPENDIX A -- THE CHARACTER SET

A-1

APPENDIX B INDEX TO SYMBOLS

B-1

APPENDIX C TABLE OF LISP 1.5 CORRESPONDENCES

C-1

APPENDIX D A COMPLETE LISP PROGRAM EXTENDING
THE OPERATORS FOR COMBINATORIAL
ANALYSIS

D-1

SECTION 1

DATA STRUCTURE OF THE LANGUAGE

LISP is a formal mathematical language designed primarily for symbolic data processing. All data in LISP are in the form of symbolic expressions called S-expressions. S-expressions are of indefinite length and have a branching tree structure. The assignment capability of B 6700/LISP permits circular, self-embedding, and lattice as well as tree structures to be built.

Computations on S-expressions are described as recursive functions or programs of the S-expressions. These programs have representations in the form of S-expressions and therefore can serve not only as descriptions of computations but also as objects of a computation. LISP can then interpret or execute programs in the form of S-expressions.

S-EXPRESSIONS

All data in LISP ultimately have representations in the form of S-expressions. The most elementary form of S-expression is the atom. Atoms have no sub-structure that is accessible to the user. In B 6700/LISP, there are two types of atoms: symbols and numbers. Symbols are atomic S-expressions that have a unique identity. Symbols can be used as tokens to represent other objects, as names to identify other S-expressions, and as carriers of their associated print image. Numbers are atomic S-expressions normally used to represent nonnegative integers less than 1,048,576, although floating point numerics are available.

Syntactically, atomic expressions correspond to lexical units of input and output. In B 6700/LISP, five categories of lexical units are distinguished. These are reserved words (RW), operators (OP), numbers (NUM), identifiers (ID), and special symbols (SP). These are defined below using Backus-Naur form (BNF). The definitions of <letter>, <digit>, <rw char>, and special char <SP> depend on the character set of the I/O terminal. The character set for the correspondence terminal using an APL typing element is used throughout this manual. The character set is shown in Appendix A. A lexical unit cannot have internal spaces.

<OP> ::= <special char>

1. <NUM> <digit>

id s. 30. | <ID> <letter>

$\Delta \text{SP} = \frac{\pi}{\lambda} (n_1 d_1 - n_2 d_2) + \left[\frac{n_1^2 - n_2^2}{2(n_1 + n_2)} \right] \sin(2\delta)$

$$\langle \text{SYMBOL} \rangle ::= \langle \text{PRIM SYMB} \rangle \mid \langle \text{SP} \rangle$$
$$\langle \text{LEX UNIT} \rangle ::= \langle \text{SYMBOL} \rangle \mid \langle \text{NUM} \rangle$$

The first four lines below each contain a single lexical unit corresponding to a single atom. All but the first are symbols. The remaining line contains 14 symbolic lexical units. Reserved words begin with a.

$$\alpha \text{IF ISATOM } (X) \alpha \text{THEN EVAL } (Y, \text{ENV}) \alpha \text{ELSE } Z$$

COMPOSITE DATA

Composite data are S-expressions formed by combining other S-expressions. There are three essential operations for this purpose: cons, car, and cdr. The operation cons(X, Y) consolidates the two S-expressions X and Y to form a single S-expression. The operations car and cdr are used respectively to recover the original X and Y used for the consolidation. Thus, there are two identities:

$$\text{car}(\text{cons}(X, Y)) = X$$

$$\text{cdr}(\text{cons}(X, Y)) = Y$$

A composite datum can be viewed as a two part container with each part containing a pointer to other data. In this view, cons(X, Y) has two parts:

TYPE OF X	CAR: X	CDR: Y
-----------------	-----------	-----------

The X and Y are pointers. The type of X is necessary because X can have arbitrary type. Y must always be a list (i.e., components), so its type can be implicit. The types are number, symbol and list.

LISTS

Composite S-expressions are usually interpreted as lists. That is, if X is any S-expression and Y is a list, then cons(X, Y) is a list whose first element is X, whose second element is the same as the first element of list Y, and where the (i+1)st element is the same as the ith element of list Y for all elements of Y. Ultimately, repetitive application of cdr to a list must yield the empty list. In B.6700/LISP there is such a list.

CONSTRUCTS

Another interpretation of S-expressions is as constructs. Here symbols are considered as names or attributes or properties designating other S-expressions as their value. Constructs are thus even length lists of which the first of each pair of elements is a symbol naming the value that is the second member of the pair. Constructs can be used to describe objects, with the names designating the attributes to be described and the corresponding value giving the value of that attribute. The list (COLOR, RED, SIZE, BIG, SHAPE, BOX) might be a construct describing a big red box.

LAMBDA EXPRESSIONS

Lambda expressions are the S-expression form of user-defined functions. The lambda expression is a three-list consisting of a) The symbol **LAMBDA**, b) A list of the formal parameter symbols (i.e., bound variable), and c) The expression for the body.

CONCRETE SYNTAX

The (concrete) syntax of S-expressions is simple. Atoms are represented by their print image. Composite S-expressions are represented by catenating their elements in order and surrounding them with a pair of parentheses. A comma may be optionally used as an element separator.

<S UNIT> ::= <PRIM LEX UNIT>	TYPE	OF	X
<S SEPARATOR> ::= <space>			

<S-EXP TAIL> ::= <S SEPARATOR> <S-EXP TAIL>

<S-EXP> ::= <S UNIT>
| (<S-EXP TAIL>

SECTION 2

THE METALANGUAGE

The metalanguage is the language in which LISP functions and programs are written. Expressions of the metalanguage will be called M-expressions and have two forms. The first form (concrete syntax) is that used externally to represent M-expressions and is composed from lexical units. The second form is the corresponding internal representation (abstract syntax) in terms of S-expressions. The importance of M-expressions is the meaning (semantics) associated with them by the LISP interpreter. Thus, in the description which follows, concrete syntax, abstract syntax, and semantics will be given for each form of M-expression.

ABSTRACT SYNTAX

Any computation can be represented as a list in which the value of the first element is an operation and the values of the remaining elements are the arguments for that operation. The elements of the list can be computations of the same form. All M-expressions in B 6700/LISP will have this form of S-expression as their concrete syntax.

INTERPRETATION

The interpreter will be described in more detail in Section 3. For now, it is sufficient to know that the following interpretation is given to the composite S-expression representation of an M-expression:

1. Each element of the list is evaluated in order from left to right.
2. The value produced by the evaluation of the first element is then interpreted as an operation that is applied to the values produced by the remaining list elements to produce the value of the entire expression.

The interpreter returns a value associated with the atom when it encounters an atomic S-expression. For numbers, that value is the number itself.

There is a single exception to the above interpretation rules. If the operation is quote (QT), it indicates that the remaining list elements are not to be evaluated. The second (nonevaluated) list element is taken as the value of the expression. Thus the evaluation of the M-expression $QT(X+Y)$ would produce as its value the S-expression that is the internal representation of the M-expression $X + Y$, namely, the list $(+, X, Y)$.

CONCRETE SYNTAX AND DESCRIPTION

The M-expression will take four general forms: atoms, functional form, infix operator form, and reserved word form. The Δ in the BNF-like equations below separates the right-hand sides of concrete and abstract syntaxes.

$\langle \text{ATOM} \rangle ::= \langle \text{ID} \rangle \quad \Delta \quad \langle \text{ID} \rangle$
 $\quad \quad \quad | \langle \text{NUM} \rangle \quad \Delta \quad \langle \text{NUM} \rangle$

Atomic M-Expressions are either identifiers or numbers. The abstract representation of an atomic M-expression is the corresponding symbol or number.

The interpretation of an atomic expression is the primitive value associated with that atom. The primitive value of a number is its numeric value.

PRIMITIVE EXPRESSIONS

$\langle \text{PRIM EXP} \rangle ::= \langle \text{ATOM} \rangle$

$\Delta \langle \text{ATOM} \rangle$

$| \langle \text{EXP} \rangle$

$\Delta \langle \text{EXP} \rangle$

$| [\langle \text{SEQ LIST} \rangle]$

$\Delta \text{cons} ('SEQ', \langle \text{SEQ LIST} \rangle)$

$| ''$

$\Delta ''$

$| \langle \text{LEX LIST} \rangle$

$\Delta \text{cons} ('QT', \text{cons} (\text{bind} (\langle \text{LEX LIST} \rangle), ''))$

$\langle \text{LEX LIST} \rangle ::= \langle \text{LEX UNIT} \rangle$

$\Delta \text{cons} (\langle \text{LEX UNIT} \rangle, '')$

$| \langle \text{LEX UNIT} \rangle \langle \text{LEX LIST} \rangle$

$\Delta \text{cons} (\langle \text{LEX UNIT} \rangle, \langle \text{LEX LIST} \rangle)$

$\langle \text{SEQ LIST} \rangle ::= \langle \text{EXP} \rangle$

$\Delta \text{cons} (\langle \text{EXP} \rangle, '')$

$| \langle \text{EXP} \rangle ; \langle \text{SEQ LIST} \rangle$

$\Delta \text{cons} (\langle \text{EXP} \rangle, \langle \text{SEQ LIST} \rangle)$

A primitive expression can be an atom or parenthesized expression, in which case it is represented by the atom or expression itself. In addition, there are three special forms of primitive expression. A bracketed list of expressions is used as a shorthand to invoke sequential control. The interpretation of the *SEQ* operation is to evaluate its arguments (usually called statements) in order from left to right and to take the value of the last evaluated parameter as the value of the entire expression. The *''* notation represents the empty list and is interpreted as itself. A quoted lexical unit is represented as a two list of *QT* and the symbol whose print image is the catenation of the quoted lex list. The quoted list evaluates to the symbol.

Examples: *x*

'x'

(5)

[*y*; *'y'*]

FUNCTIONAL FORM EXPRESSIONS

<FF EXP> ::= <PRIM EXP>

Δ <PRIM EXP>

| <SUB EXP> (<PARA LIST>) Δ cons (<SUB EXP>, <PARA LIST>)

<SUB EXP> ::= <FF EXP>

Δ <FF EXP>

| <SUB EXP> [<SEQ LIST>] Δ cons ('SUBSCRIPT', cons

(<SUB EXP>, <SEQ LIST>))

<PARA LIST> ::= <EXP>

Δ cons (<EXP>, NIL)

| <EXP>, <PARA LIST>

Δ cons (<EXP>, <PARA LIST>)

A functional form expression is either a primitive expression or a prefix functional form with the list of actual parameter expressions enclosed in parentheses. Subscripted expressions are formed by enclosing a list of the expressions for the subscripts in brackets to the right of the expression whose value is to be subscripted. Internally, the subscript expression gets converted to an S-expression equivalent to the functional form operator *SUBSCRIPT*. The values of the following three examples will all be the same, but only the first two have the same representations as S-expressions.

Examples: X [2: 3]

SUBSCRIPT(X, 2, 3)

X[2][3]

F ()

(X)

(2)

(1, 1)

INFIX EXPRESSIONS

<INFIX EXP> ::= <SUB EXP>

Δ <SUB EXP>

| <OP> <INFIX EXP>

Δ cons (<OP>, cons (<INFIX EXP>, ' '))

| <SUB EXP> <OP> <INFIX EXP>

Δ cons (<OP>, cons (<SUB EXP>, cons (<INFIX EXP>, ' ')))

| <FF EXP> + <INFIX EXP>

Δ cons ('+', cons (cons ('QT', cons (<FF EXP>, ' ')), cons (<INFIX EXP>, ' ')))

| <SUB EXP> [<EXP>] <SUB TAIL>

Δ cons ('ASNSUBSCRIPT', cons (<SUB EXP>, cons (<EXP>, <SUB TAIL>)))

<SUB TAIL> ::= 1 + <INFIX EXP>

Δ cons (<INFIX EXP>, ' ')

; <EXP> <SUB TAIL>

Δ cons (<EXP>, <SUB TAIL>)

Infix expressions permit the use of unary preoperators and binary infix operators. All infix expressions are parsed right recursively without precedence of operators. Assignment using the + operator is a special case because there is an implied quoting of the first parameter. The ASNSUBSCRIPT operator can be denoted by assignment (+) to a subscripted expression. The operator := is equivalent to +.

Examples:

I-3

.NOT X

X[2]+5

RESERVED WORD EXPRESSIONS

$\langle \text{RW EXP} \rangle ::= \langle \text{INFIX EXP} \rangle$

$\Delta \langle \text{INFIX EXP} \rangle$

$| \langle \text{RW LIST} \rangle$

$\Delta \text{cons}(\text{bind}(\text{car}(\langle \text{RW LIST} \rangle)), \text{cdr}(\langle \text{RW LIST} \rangle))$

$\langle \text{RW LIST} \rangle ::= \langle \text{RW} \rangle \langle \text{RW EXP} \rangle$

$\Delta \text{cons}(\text{cons}(\langle \text{RW} \rangle, ' '),$

$\text{cons}(\text{cons}('QT', \text{cons}(\langle \text{RW EXP} \rangle, ' ')), ' '))$

$| \langle \text{RW} \rangle \langle \text{INFIX EXP} \rangle \langle \text{RW LIST} \rangle$

$\Delta \text{cons}(\text{cons}(\langle \text{RW} \rangle, \text{car}(\langle \text{RW LIST} \rangle)),$

$\text{cons}(\text{cons}('QT', \text{cons}(\langle \text{INFIX EXP} \rangle, ' ')),$

$\text{cdr}(\langle \text{RW LIST} \rangle))$

The reserved word form of expression permits quoting in the abstract syntax without explicit use of the QT in the concrete syntax. This is particularly useful for control operations. A reserved word expression consists of a sequence of reserved words, each followed by an actual parameter expression. Each of the parameter expressions will be quoted in the abstract syntax. The operation symbol is formed by "binding" the sequence of reserved words. Binding is a means of forming a single unique symbol from a list of symbols and will be explained further in Section 3.

Examples:

$\text{aWHILE } X \text{ aDO } Y$

$\text{aIF } C \text{ aTHEN } X \text{ aELSE } Y$

The operator UQT can be used to inhibit any implicit quoting in the M-language. Thus the expression $X+3$ would be interpreted as an assignment of the numeric value 3 to variable X , but $UQT(X)+3$ would cause 3 to be assigned to the variable whose name is the value of X .

PROGRAMS

<PROG> ::= <EXP>

Δ <EXP>

<EXP> ::= <RW EXP>

Δ <RW EXP>

Any form of expression is an M-expression. A program is a single M-expression. This is possible and convenient because 1) All operations including declarations are executable; 2) The *SEQ* operation (denoted with brackets) provides a program feature for executing sequences of expressions, 3) Recursive and iterative controls allow expansion of computation without a proportional expansion in the program description, and 4) Global variables retain their value between programs.

Examples:

```
[Y + '1'; αWHILE X .NEQ '1' αDO [Y + CONS(CAR(X), Y); X + CDR(X) ]]
```

```
X[3] + Y + 3 * X[5]
```


SECTION 3 THE OPERATION SET

In this section, the operations available in B 6700/LISP are described. The syntax and interpretation is given for each operation. In many cases, a formal description as a user-defined operation is given along with the natural language description.

DEFINITION OF RECURSIVE FUNCTIONS AND SYNTAX EXTENSIONS

α OP FORM α IS BODY

The α OP α IS operation defines new operations. The FORM gives the M-expression representation of the operation being defined. Parameter positions are occupied by identifiers that act as the bound variables or formal parameters of the body. The BODY is an expression that defines the function of the operation in FORM. The α OP α IS operation forms the corresponding LAMBDA expression and assigns it to the defined name.

The FORM can be a reserved word expression, functional form expression, a unary operator expression, or binary operator expression. The formal parameter identifiers of a FORM can be preceded by a colon (:) to indicate a variable number of parameters. Actual parameter values are associated with non-colon formal parameter names on a one-for-one basis in the same order from either or both ends (although they are evaluated from left to right). All remaining actual parameters are formed into a list that is associated with the colon parameter.

Examples of forms:

α WHILE C α DO X

CONS(X, :Y)

.NOT X

X + Y

ELEMENTARY LIST PROCESSING OPERATIONS

The elementary list processing operations are those required for manipulation of list structured data. These operations can be divided into three categories: constructors, selectors, and predicates. The constructor operation *CONS* is used to build new data. The selectors *CAR* and *CDR* are used to extract parts of previously constructed data.

Predicates are functions whose value is a truth value. Numbers are used as truth values here with zero interpreted as false and all other values as true. The initial values of the identifiers *TRUE* and *FALSE* are 1 and 0, respectively. *NIL* is the empty list.

CONS(X, :Y)

The *CONS* operation is a generalization of the cons function introduced in Section 1. *CONS* of a single argument is the identity operation. *CONS* of two arguments is identical to cons. Otherwise *CONS(X, :Y)* is cons(*X*, *Z*) where *Z* is the *CONS* of the elements of *Y*. Formally:

$\alpha \text{OP } \text{CONS}(X, :Y) \alpha \text{IS } \alpha \text{IF } Y = \text{NIL } \alpha \text{THEN } X \alpha \text{ELSE}$
 $\text{cons}(X, \text{APPLY}(\text{CONS}, Y))$

CAR(X)

This operation returns the first part of the composite *X*. That is, *CAR(CONS(X, :Y))* is *X*.

CDR(X)

The *cdr* operation returns the second part of the composite *X*. That is, *CDR(CONS(X, :Y))* is *CONS(:Y)*.

X = Y

X=Y is a predicate that tests the values of *X* and *Y* for identity. Thus, any two instances of the same number are equal under *=*, as are the values produced by two successive references to the same variable. The expression *CONS(X, Y) = CONS(X, Y)*, however, yields false because the two lists are distinct, even though they have the same structure and components. The symbol *.EQ* may be used instead of *=*.

$X \neq Y$

The predicate $\neq$ is true if and only if $X = Y$ is false. The operator $\neq$ is equivalent to $\neq$.

$\alpha OP X \neq Y \alpha IS$

$\alpha IF X = Y \alpha THEN FALSE \alpha ELSE TRUE$

$\alpha OR O = X = Y$

$ISATOM(X)$

The $ISATOM$ predicate is true if and only if X is an atom.

$ISSYMB(X)$

The $ISSYMB$ predicate is true if and only if X is a symbol.

$ISLIST(X)$

The $ISLIST$ predicate is true if and only if X is a list.

$ISNUM(X)$

The $ISNUM$ predicate is true if and only if X is numeric.

$APPEND(X, Y)$

The $APPEND$ operation combines its two arguments into one new list.

$CONC(:X)$

$CONC$ catenates (concatenates) its arguments at the top level without copying.

MEMBER (X,Y)

This predicate is true if and only if X is a member of list Y .

REVERSE (X)

This operation builds a list with the elements of list X in reverse order.

SEARCH (X,P,F,U)

This function searches the list X for an element that has property P . If found F of the value is returned. Otherwise $U(NIL)$ is returned.

α OP SEARCH(X,P,F,U) α IS

[α WHILE (X .NEQ NIL) .AND .NOT P(CAR(X)) α DO'X:= $\dot{C}$ DR(X);

α IF X=NIL α THEN U(NIL) α ELSE F(CAR(X))]

SUBLIS(X,Y)

For each attribute of construct X substitute its value for its name in Y .

SUBST(X,Y,Z)

Substitute X for all occurrences of Y in Z .

LIST (:X)

Form a list from the elements :X.

SYMBOL ()

Create a new unique symbol with α and an identifying integer as its print image.

SYMBOL (X)

Create a new unique symbol with the same print image as symbol X .

CONSTRUCT PROCESSING AND ASSIGNMENT

REF(X, Y)

If X is an integer, then $REF(X, Y)$ yields the x th element* of the list Y . If X is a symbol, the $REF(X, Y)$ yields the value associated with the name X in construct Y .

ASN(X, Y, V)

If X is an integer, then $ASN(X, Y, V)$ replaces the x th element* of list Y by the value V . If X is a symbol, the $ASN(X, Y, V)$ replaces the value associated with the name X in construct Y by value V .

ASNCDR(X, V)

This operation replaces the cdr of list X by the list V . In effect all occurrences of X in the system are replaced by the list:

$CONS(CAR(X), V)$

RHO(X)

This operation returns an integer that is the number of elements in list X .

$\alpha OP RHO(X) \alpha IS$

$\alpha IF X = ' ' \alpha THEN 0 \alpha ELSE RHO(CDR(X)) + 1.$

ISSAME(X, Y)

This predicate is true if and only if X and Y are identical or if elements of lists X and Y are pair-wise identical.

$\alpha OP ISSAME(X, Y) \alpha IS$

$\alpha IF X = Y \alpha THEN TRUE \alpha ELSE$

$\alpha IF ISATOM(X) . OR ISATOM(Y) \alpha THEN FALSE \alpha ELSE$

$\alpha IF CAR(X) = CAR(Y) \alpha THEN ISSAME(CDR(X), CDR(Y))$

$\alpha ELSE FALSE$

* Origin 0 is assumed.

COPYLIST(X)

COPYLIST returns a copy of X , that is, a list having the same elements as list X .
COPYLIST of an atom is an identity operation.

601 **COPYLIST(X)** **ars**

αIF ISATOM(X) αTHEN X αELSE

αIF X = ' ' αTHEN X αELSE

CONS(CAR(X), COPYLIST(CDR(X)))

COPY(X)

This operation makes a copy of the list structure X . That is, all sublists are also copied.

EFFACE(X,Y)

This operation deletes the first appearance of the item X from the list Y .

PAIR(X,Y)

This operation pairs the attributes on list X with the values on list Y to form a new construct.

PROP(X,Y)

This operation returns the rest of list Y beyond the first appearance of X .

REMPROP(X,Y)

This operation deletes the attribute X and its value from the construct Y .

ENVIRONMENTAL, SCOPE AND DEFINITION OPERATIONS

Expressions are evaluated in the context of some environment. The environment determines what value is to be associated with the atomic elements of the expression. For the αOP αIS operation, the most local environment of the body contains the formal parameter names with the corresponding actual parameter values as their associated values.

$\alpha SCOPE$ X

The $\alpha SCOPE$ operation delineates the scope of definitions. Definitions made within the expression X are not defined after the evaluation of X . The body of an αOP αIS has an implicit scope.

$DEF(:X, V)$

The DEF operation defines new local variables $:X$ to have initial value V . The definitions mask the more global definitions of the same names. These masked global definitions of the names $:X$ again become effective upon exit from the most local scope.

X

The lone occurrence of an identifier is interpreted as a reference on the most local definition of that identifier.

$PREF(X)$

The primitive reference operation $PREF$ references the value of the variable whose name is the value of X .

$X \leftarrow V$

The assignment operator $\leftarrow$ replaces the value associated with the most local definition of X by the value V . In no case will the operator fail; if the user attempts to store a value into a previously undefined identifier, the operator drops into a definition mode. The variable is added to the most global environment, and the assignment is completed. There is an implied quote on X . Global variables can be undefined by assigning NIL . The symbol $:=$ can be used instead of $\leftarrow$.

PASN(X, V)

The primitive assignment operation **PASN** assigns the value of **V** to the variable named by the value of **X**.

Y[:X]

The subscript operation is used to subscript (i.e., reference a component) a list or construct **Y**. The subscripts **:X** are applied successively to the preceding result using the operator **REF** and beginning with **Y**. This operation is also called **SUBSCRIPT**.

αOP Y[:X] αIS

[αWHILE X ≠ ' ' αDO

[Y ← REF(CAR(X), Y) ; X ← CDR(X)] ; Y]

Y[:X, Z] ← V

The assign to subscript operation changes the value associated to a subscript position. The new value of **Y[:X, Z]** will be **V**. This operation is also called **ASNSUBSCRIPT**.

αOP Y[:X, Z] ← V αIS

[Y ← REF(CAR(X), Y) ; X ← CDR(X)] ;

ASM(Z, Y, V)]

ITERATIVE AND CONDITIONAL CONTROL

αIF C αTHEN X αELSE Y

The αIF αTHEN αELSE form is the primitive conditional operation. Either X or Y is evaluated, depending on whether the evaluation of C is true or false, respectively.

αIF C αTHEN X

The αIF αTHEN form of conditional is equivalent to αIF C αTHEN X αELSE ' ' and is normally used as a statement.

[:X ; Y]

This form is used for sequential control. The :X's are evaluated from left to right. Y is then evaluated and taken as the value of the entire expression. This operation is also called SEQ.

αOP [:X ; Y] αIS Y

The quote operation inhibits the evaluation of its argument. The evaluation of QT(X) yields the expression X. Syntactically X can be any expression or it can be any single symbol.

UQT(X)

The unquote operation enables the evaluation of its argument. Its use is primarily to override the effect of implied QT operations in reserved word and assignment operations.

αWHILE C αDO X

The αWHILE αDO operation is an iterative control operation that repetitively evaluates the expression X as long as the evaluation of C produces true. C is evaluated before each iteration. The last value of X is returned.

αDO X αWHILE C

The **αDO αWHILE** operation is similar to **αWHILE αDO** except that **C** is evaluated after each iteration.

αOP (αDO X αWHILE C) αIS

[DEF('T', EVAL(X));

αWHILE EVAL(C) αDO EVAL(X); T]

QTL(X)

The **QTL** operation quotes each of its arguments and forms them into a list in the order given. None of the arguments are evaluated.

CODE(X)

This forms the internal operation code with index **X**.

FIRST(X,:Y)

This operation is similar to **SEQ** except the first value **X** is returned as the value.

αPROG X

This operation permits **αGOTO** transfers of control within the **SEQ** expression. Labels appear as atomic arguments to **SEQ**.

αGOTO X

This operation causes an unconditional transfer of control to label **X** within the most local **αPROG**.

SELECT(Q,:X)

This operation evaluates every other element sequentially from the first of **:X** until one is found whose value is **Q**. It returns the value of the next element of **X**. If not found, it returns **NIL**.

MAPLIST(X,F)

This operation forms a list of the values resulting from applying function **F** to the elements of the list **X**.

ARITHMETIC, RELATIONAL, AND BOOLEAN OPERATIONS

The following arithmetic operations are provided. In most cases these are full word floating point operations. For a full description see the Burroughs B 6700 Extended Algol Reference Manual.

$X + Y$	add	$EXP(X)$	exponential
$X - Y$	subtract	$FIRSTONE(X)$	bit number of first one bit
$-X$	negate	$GAMMA(X)$	gamma function
$X \times Y$	multiply	$LN(X)$	natural logarithm
X / Y	divide	$LNCGAMMA(X)$	natural log of gamma function
$X \div Y$	integer divide	$LOG(X)$	logarithm base 10
$X \mid Y$	residue	$MAX(:X)$	maximum
$ABS(X)$	absolute value	$MIN(:X)$	minimum
$AND(:X)$	full word and	$X .MPY Y$	multiply
$ARCCOS(X)$	arc cosine	$NABS(X)$	negative absolute value
$ARCSIN(X)$	arc sine	$ONES(X)$	number of one bits
$ARCTAN(X)$	arc tangent	$OR(:X)$	full word inclusive or
$ARCTAN2(X, Y)$	arc tangent	$PWD(X, S, L)$	end round partial word
	divide	$X .RES Y$	residue
$COS(X)$	cosine	$X .RTP Y$	raise to power
$COSH(X)$	hyperbolic cosine	$SIGN(X)$	sign
$COTAN(X)$	cotangent	$SIN(X)$	sine
$X .DIV Y$	integer division	$SINH(X)$	hyperbolic sine
$ENTIER(X)$	entier	$SQRT(X)$	square root
$ERF(X)$	standard error function	$TAN(X)$	tangent
		$TANH(X)$	hyperbolic tangent
		$XOR(:X)$	full word exclusive or

The following relational operations are provided. These operations apply only to numeric values.

$X = Y$	equal	$X .GEQ Y$	greater or equal
$X \neq Y$	not equal	$X .GTR Y$	greater than
$X > Y$	greater than	$X .LEQ Y$	less or equal
$X < Y$	less than	$X .LSS Y$	less than
$X .EQ Y$	equal	$X .NEQ Y$	not equal

The following Boolean operations are provided.

$X .AND Y$

$.NOT Y$

$X .OR Y$

THE UNIVERSAL LISP FUNCTION

The universal LISP function is that function or operation that can compute the value of any LISP expression in the context of any given environment. Thus the universal LISP function called `EVAL` is in fact an interpreter for the LISP language.

`EVAL(X, ENV)`

The LISP interpreter operation `EVAL` evaluates the expression `X` in the context of environment `ENV`. The expression `X` is any S-expression representation of a LISP program (i.e., expression). The environment `ENV` is a construct that describes the environment in which `X` is to be evaluated. The defined identifiers of the environment are paired with their value in `ENV`. Identifiers of `X` that are not defined by `ENV` are defined by the current environment at the time of call on `EVAL`.

`EVAL(X)`

The interpreter can be called with a single argument. `EVAL(X)` evaluates the expression `X` in the context of the environment of the call on `EVAL`.

`APPLY(F, X)`

The operation `APPLY` applies the operation `F` to the list of actual parameter values of `X`. Thus, for any triadic operation `G` and any expressions `A`, `B`, and `C`, the following two forms are equivalent:

1. `G(A,B,C)`
2. `APPLY (G, CONS(A,B,C, ' '))`.

This operation is equivalent to the LISP 1.5 function `EVALQUOTE (F, X)`.

INPUT-OUTPUT OPERATIONS

Inputs and outputs are normally in the form of S-expressions. Inputs may also be as a list of lexical units. LISP programs can be input in the form of M-expressions. In the latter case no explicit operation is used. The normal sequence of actions with the LISP system provides for the input of a program in the form of an M-expression, translation of the expression into the corresponding S-expression, interpretation of the S-expression and output of the resulting value as an S-expression.

INPUT()

The **INPUT** operation reads one input line as a list of lexical units (i.e., atoms) and returns that list as its value.

READSEXP()

The **READSEXP** operation reads the next input S-expression and returns that expression. The S-expression can occupy multiple lines but cannot share a line with other inputs.

OUTPUT(X)

The **OUTPUT** operation displays the S-expression *X*. The value returned by this operation is *X* itself.

BIND(X)

It is sometimes desirable to have a single symbol to stand for a list of symbols. The **BIND** operation binds the list *X* into a single symbol. If **ISSAME(X, Y)** then **BIND(X) = BIND(Y)**. This operation is used by the LISP translator to combine reserve words into a single operation symbol. On output there is no difference between the image of *X* and the image of **BIND(X)**.

UNBIND(X)

The operation **UNBIND** returns a list of the elements that were bound to form the symbol *X*.

SET(DECPLAC,VAL)

This operation sets the number of decimal places to be printed when noninteger numeric values are output. The previous value is returned. This, and the following set operations and those described later have a first argument that identifies the internal switch or variable to be set or read.

SET(DISKINPUT,VAL)

This operation sets a Boolean condition controlling input from a symbolic disk file. When set input is taken from the disk file named in the variable *DISKFILE*. An end of file from disk will turn off *DISKINPUT*. The previous value is returned.

SET(WIDTH,VAL)

This operation sets the character width for both input and output. The proper setting is determined by the terminal. The previous value of *WIDTH* is returned.

OUTPUT(X)

The *OUTPUT* operation displays the S-expression *X*. This operation is a list.

RIMP(X)

It is sometimes desirable to have a single symbol to stand for a list of symbols. The *RIMP* operation binds the list *X* into a single symbol. If *RENAME* is a symbol, *RIMP(X) = RENAME(X)*. This operation is used by the LISP translator to convert reserve words into a single operation symbol. On output, there is no difference between the image of *X* and the image of *RIMP(X)*.

UNBIND(X)

The operation *UNBIND* returns a list of the elements that were bound to the symbol *X*.

CONVERSATIONAL AND DEBUGGING FACILITIES

Expressions can be executed one at a time from the terminal. This is in fact the only mode of operation, since all programs are expressions. There are, however, facilities that allow continuity between expressions entered one at a time. The global environment contains all predefined identifiers and all identifiers that are assigned in programs but not explicitly defined. This global environment remains with the LISP workspace and can be modified conversationally.

Continuity from run to run can be maintained by saving workspaces on disk. Each LISP user has a library of workspaces. The B 6700 disk file name is formed by adding preidentifiers to the workspace name so that the actual workspace file name is *LISP/user/name*, where *user* is the value associated with the identifier *USER* and *name* is the workspace name specified in the workspace operations. Workspace and user names must be identifiers.

CREATE(WSID)

The *CREATE* operation creates a new workspace file with the name *WSID*. The current workspace is copied into that file. The operation will take place only if there were no previous file by the same name for this LISP user. *CREATE* is a predicate that returns true if and only if the operation is successful.

SAVE(WSID)

The *SAVE* operation is a predicate that saves the current (active) workspace under file name *WSID*. It will override any previous file by the same name or create a new file.

LOAD(WSID)

The *LOAD* operation is a predicate that loads the designated workspace file over the current workspace without saving it.

ERASE(WSID)

The *ERASE* operation is a predicate that removes the workspace file named by *WSID* from disk.

SLOAD(NAME)

The system load operation **SLOAD** is a predicate which loads special system workspaces. The name **CLEAR** is used to designate the normal initialized workspace.

OPVARS()

The **OPVARS** operation returns a two-list of lists. The first is a list of the currently defined operator symbols. The second is a list of all other defined symbols.

TRANSLATE(X)

This operation interprets the list **X** as a string of lexical units in the form of an M-expression. This string is translated into the corresponding S-expression and returned as value.

SET(TRACE, V)

This form of the **SET** operation enables (or disables) tracing if the Boolean value of **V** is true (false). When trace is on the value at each step of the most global sequential operation is output. The previous value is returned.

SET(TREE, V)

This form of the **SET** operation enables (or disables) tracing if the Boolean value of **V** is true (false). When trace is on the value at each step of the most global sequential operation is output. The previous value is returned.

CLEARWS()

This operation initializes the workspace and is equivalent to **SLOAD('CLEAR')**.

CLEARSTACK()

The **CLEARSTACK** operation jumps out all levels of execution and returns control to input. The message **CLEARSTACK** is printed.

END()

This operation is equivalent to an end of file from cards. It is the normal way of terminating a program.

PAUSE()

This operation causes execution to halt until input from the terminal. The message **PAUSE** is printed. If the input is 0, then the stack is cleared. Otherwise processing continues.

αTRACE X

This operation causes subsequent execution of the operation **X** to print **TRACE** its name, and the resulting value upon exit.

UNTRACE(X)

This operation turns off tracing of operation **X**.

SET(X)

The unary **SET** operation is used to interrogate the setting of several internal LISP switches and variables. The values of **X** follow (* indicates it can be set):

CPUTIME	Processor time in 1/60 sec.
CYCLES	Total number of interpretation cycles
DECPLAC	Number of decimal digits to print *
DISKINPUT	Disk input switch *
GOLPRINT	Implicit garbage collection print switch *
IOTIME	I/O time in 1/60 sec.
LEXSPACE	Unused lexspace *
LISTSPACE	Unused list space *
REALSPACE	Unused real number space *
SPLIMINC	Increment on list space at garbage collection *
STACKSPACE	Unused stack space
TIMEOFDAY	Time of day in 1/60 sec.
TRACE	Sequential trace switch *
TREE	Print parse tree switch *
WIDTH	Character width for input and output. *

normal way

The
Otherwise

CE

al
se

SECTION 4 MEMORY ALLOCATION AND GARBAGE COLLECTION

The LISP system dynamically allocates memory one word at a time from a large linear segmented array. Additional space is required when new composite data are created (e.g., by `CONS`). Not only are new data objects built but also data structures are discarded. If, for example, a variable `X` contains the only reference to a datum `Y` and the content of `X` is replaced, there are no remaining references to `Y`. Thus `Y` could be restored to the pool of available space. The process of examining the list memory, determining which words are no longer accessible, and returning these words to available space is called garbage collection.

Garbage collection is normally called for implicitly when a prescribed amount of memory has been allocated. This amount is called `SPLIMINC` for space limit increment and can be set by the user. The value of `SPLIMINC` is actually a lower bound on the amount of space allocated between two consecutive garbage collections and has no direct relation to the total amount of space allocated. The amount of space in use after garbage collection is dependent only on the amount of data required at that point in the program. The amount allocated between consecutive garbage collections is determined wholly by `SPLIMINC`. If `SPLIMINC` is too small, garbage collection will occur frequently; and the program will require excessive processor time. If `SPLIMINC` is too large, garbage collection will be infrequent and memory will be used by collectable data, thus increasing memory requirements, increasing memory swapping, and reducing response times. Garbage collection will also be called for when the amount of space remaining for stack, real number, or lexical table is exhausted.

Increment on list space
Used stack space
Time of day in 1/60 sec
Second last free switch
First last free switch
Character with for input and output

SPACE
TIME
INPUT

The following LISP operations are provided to aid the user in controlling garbage collection:

GCL ()

The **GCL** operation enables the user to explicitly call for garbage collection at any point in the program. The value returned is the number of words recovered.

SET (SPLIMINC, VAL)

This form of the **SET** operation allows the user to limit the size of **SPLIMINC**. The best value for **VAL** depends on the hardware configuration, the particular program, and the kind of service desired.

SET (SPACE, VAL)

The **SET** operation can also be used to control the amount of space remaining in the system. The argument **SPACE** can be **LISTSPACE**, **STACKSPACE**, **REALSPACE**, or **LEXSPACE**. The previous amount of space remaining is returned.

SET (GCLPRINT, VAL)

This operation sets a Boolean that controls printing of the message **GARBAGE COLLECTION** and the amount of list space recovered when implicit garbage collections occur.

garbage

SECTION 5

HOW TO USE THE SYSTEM

The system may be used from a typewriter terminal using an APL element; from any CANDE terminal including TC 500, BIDS, CONRAC, and TTY; from teletype-compatible terminals; or from a card reader and printer. The label equate in the execute or run command determines the type of device to be used. The file identifiers are *APLE* for the APL element, *CANDE* for the CANDE terminal, *TTY* for the teletype-compatible terminal, and *CARD* for the card reader.

The program is activated by one of the following commands. The first two forms are for use with *CANDE* with and without the *SITE* option, respectively. The third is for non-*CANDE* terminals, and the fourth is for card input.

E LISP/SITE (fileid)

E *SYSTEM/LISP (fileid)

?RUN SYSTEM/LISP (fileid)

?EXECUTE SYSTEM/LISP
?fileid

The LISP system has been entered when the terminal responds with a line containing only:

LISP

The next line of output should be:

CLEAR

This indicates that the clear workspace has been loaded. If the output is 0 instead of **CLEAR**, there was a failure in the attempt at loading the clear workspace.

The keyboard will then be unlocked for input by the user. Each unit of input must be an expression and must occupy an integral number of lines. A unit of input is terminated as the first line that forms a complete M-expression. Thus, if the expression $X+5$ were to be entered, the $X+$ could be entered on the first line and the 5 on the second line. If, however, the X were typed on a line by itself, it would be taken as a complete expression; and the $+5$ would be the next expression.

Expressions are executed one at a time as they are entered. These expressions can modify the content of the global environment of the current workspace. The conversational mode of operation is useful for debugging and checkout. For longer programs including those entered from cards, the program should be enclosed in sequential brackets, thus ensuring that the entire program will be interpreted as a single expression.

The response from an expression entered from the terminal is the value of that expression in S-expression form. If, after a reasonable amount of time, no response is obtained, enter **],)** or **'** to complete the last entered M-expression.

There are several error conditions that can occur in LISP programs. Some error conditions of the LISP system terminate the last entered expression and return control to the user. He may then enter another expression. Other errors are printed, and control is immediately returned to the program with value **NIL**. Many common errors, however, have not been implemented as LISP system errors. Instead the operations in which the error occurs has been defined as predicates that produce **FALSE** in case of error, (e.g., the workspace operations). In other cases, the operation produces **NIL** as its value (e.g., **REF(X, Y)** yields **NIL** if X is not defined in Y). In many cases additional diagnostic information is given with the error message.

The error messages of the LISP system follow:

- | | |
|-------------------------------|---|
|)ILLEGAL OPERATION - | Indicates that the value of an operator expression was neither a primitive operation nor a lambda expression. |
|)TOO FEW PARAMETERS - | Indicates that an operation requires more actual parameters than were provided. |
|)TOO MANY PARAMETERS - | Indicates that an operation requires fewer actual parameters than were provided. |

)TYPE ERROR -

Indicates that an operation was called with an actual parameter whose type was unacceptable to that operation.

)SYNTAX ERROR -

Indicates that there was a syntax error in the last entered M-expression. The translator does not terminate at the first syntax but attempts to translate the remainder of the input string as an M-expression.

)MEMORY OVERFLOW -

Indicates that list, stack, real, or lexical space is exhausted after garbage collection.

)END OF FILE -

Indicates end of file on input.

)MISSING RIGHT QUOTE -

Indicates that an end-of-line has been entered in the middle of a quoted string. The string may continue on the next line.

)ILLEGAL FORMAL PARAMETER -

Indicates that the formal parameter list part of a lambda expression is not of proper form.

THE CHARACTER SET

...the definition of the syntactic categories <letter> and <special char> as used in Section 1.

APPENDIX A

THE CHARACTER SET

The particular characters available depend on the input device used. The table below gives the character sets when a correspondence terminal with an APL typing element is used and when a card reader is used. When the APL element is used special characters may be overstruck to form new characters and the character `^` causes the remainder of any input line to be ignored. When any other terminal (or card reader) is used, the character `%` causes the remainder of the input line to be ignored.

The character set affects the definition of the syntactic categories `<letter>`, `<digit>`, `<rw char>`, and `<special char>` as used in Section 1.

	<u>APL Element</u>	<u>EBCDIC Card Input</u>
<code>< letter ></code>	ABCDEFGHIJKLM NOPQRSTUVWXYZ <u>ABCDEFGHIJKLM</u> <u>NOPQRSTUVWXYZ</u>	ABCDEFGHIJKLM NOPQRSTUVWXYZ
<code>< digit ></code>	0123456789	0123456789
<code>< rw char ></code>	<code>^</code>	<code>\$</code>
<code>< other chars ></code>	[] () , ; + ' .	[] () , ; # " .
<code>< special char ></code>	+ - / < > * : =	+ - / < > * : =
	x = \ " ^	@ % & + #
	- v ÷ ? a e p ~	
	† ‡ † † † † † †	

V-A-0-1-2-3-4-5-6-7-8-9

1 T-1

APPENDIX B

INDEX TO SYMBOLS

Symbol on Form	Type
3-10	010
3-11	010
3-12	030
3-13	040
3-14	050
3-15	060
3-16	070
3-17	080
3-18	090
3-19	100
3-20	110
3-21	120
3-22	130
3-23	140
3-24	150
3-25	160
3-26	170
3-27	180
3-28	190
3-29	200
3-30	210
3-31	220
3-32	230
3-33	240
3-34	250
3-35	260
3-36	270
3-37	280
3-38	290
3-39	300
3-40	310
3-41	320
3-42	330
3-43	340
3-44	350
3-45	360
3-46	370
3-47	380
3-48	390
3-49	400
3-50	410
3-51	420
3-52	430
3-53	440
3-54	450
3-55	460
3-56	470
3-57	480
3-58	490
3-59	500
3-60	510
3-61	520
3-62	530
3-63	540
3-64	550
3-65	560
3-66	570
3-67	580
3-68	590
3-69	600
3-70	610
3-71	620
3-72	630
3-73	640
3-74	650
3-75	660
3-76	670
3-77	680
3-78	690
3-79	700
3-80	710
3-81	720
3-82	730
3-83	740
3-84	750
3-85	760
3-86	770
3-87	780
3-88	790
3-89	800
3-90	810
3-91	820
3-92	830
3-93	840
3-94	850
3-95	860
3-96	870
3-97	880
3-98	890
3-99	900
4-00	910
4-01	920
4-02	930
4-03	940
4-04	950
4-05	960
4-06	970
4-07	980
4-08	990
4-09	1000

APPENDIX B

INDEX TO SYMBOLS

In the table below, *VAR* identifies a symbol used as a variable, *SET_n* identifies a symbol used as the first argument to *SET* and gives its integral value, *LAMBDA* identifies an operation which is defined by a lambda expression, and *NN_c* identifies operations defined by code and gives their internal index.

<u>Symbol and Form</u>	<u>Type</u>	<u>Page</u>
<i>ABS(X)</i>	61c	3-11
<i>AND(:X)</i>	81c	3-11
<i>X AND Y</i>	39c	3-12
<i>APPLE</i>	VAR	5-1
<i>APPEND(X,Y)</i>	LAMBDA	3-3
<i>APPLY(F,X)</i>	13c	3-13
<i>ARCCOS(X)</i>	65c	3-11
<i>ARCSIN(X)</i>	86c	3-11
<i>ARCTAN(X)</i>	64c	3-11
<i>ARCTAN2(X,Y)</i>	58c	3-11
<i>ASN(X,Y,V)</i>	01c	3-5
<i>ASNCDR(X,V)</i>	43c	3-5
<i>ASNSUBSCRIPT(Y,X,:Z,V)</i>	05c	3-8
<i>BIND(X)</i>	31c	3-14
<i>CANDE</i>	VAR	5-1
<i>CAR(X)</i>	41c	3-2
<i>CARD</i>	VAR	5-1
<i>CDR(X)</i>	42c	3-2
<i>CLEAR</i>	VAR	3-17, 5-2
<i>CLEARSTACK()</i>	79c	3-17
<i>CLEARWS()</i>	30c	3-17
<i>CODE(N)</i>	47c	3-17
<i>CONC(:X)</i>	LAMBDA	3-3
<i>CONS(X,:Y)</i>	34c	3-2
<i>COPY(X)</i>	LAMBDA	3-6
<i>COPYLIST(X)</i>	11c	3-6
<i>COS(X)</i>	69c	3-11
<i>COSH(X)</i>	71c	3-11
<i>COTAN(X)</i>	67c	3-11
<i>CPUTIME</i>	SET15	3-18

Symbol and Form	Type	Page
CREATE(WSID)	50c	3-16
CYCLES	SET13	3-18
DECPLAC	SET4	3-15, 3-18
DEF(:X,V)	08c	3-7
DISKFILE	VAR	3-15
DISKINPUT	SET8	3-15, 3-18
X .DIV Y	37c	3-11
αDO X αWHILE C	20c	3-10
EFFACE(X,Y)	LAMBDA	3-6
END()	84c	3-18
ENTIER(X)	87c	3-11
X .EQ Y	26c	3-2, 3-12
ERASE(WSID)	49c	3-16
ERF(X)	76c	3-11
FVAL(X)	15c	3-13
EVAL(X,ENV)	15c	3-13
EXP(X)	75c	3-11
FALSE	VAR	3-2
FIRST(X,:Y)	LAMBDA	3-10
FIRSTONE(X)	88c	3-11
GAMMA(X)	89c	3-11
GCL()	33c	4-2
GCLPRINT	SET12	3-18, 4-2
X .GEQ Y	LAMBDA	3-12
αGOTO X	LAMBDA	3-10
X .GTR Y	28c	3-12
αIF C αTHEN X	16c	3-9
αIF C αTHEN X αELSE Y	14c	3-9
INPUT()	45c	3-14
IOTIME	SET16	3-18
ISATOM(X)	22c	3-3
ISLIST(X)	23c	3-3
ISMEMBER(X,Y)	LAMBDA	3-4
ISNUM(X)	25c	3-3
ISSAME(X,Y)	09c	3-5
ISSYMB(X)	24c	3-3
LAMBDA	VAR	1-4, 3-1
X .LEQ Y	LAMBDA	3-12
LEXSPACE	SET9	3-18, 4-2
LISP	VAR	3-16, 5-1
LIST(:X)	LAMBDA	3-4
LISTSPACE	SET3	3-18, 4-2
LN(X)	74c	3-11
LNGAMMA(X)	77c	3-11
LOAD(WSID)	53c	3-16
LOG(X)	73c	3-11
X .LSS Y	LAMBDA	3-12

Symbol and Form

Type

Page

MAPLIST(X,F)

LAMBDA

3-10

MAX(:X)

59c

3-11

MIN(:X)

60c

3-11

X.MPY Y

39c

3-11

NABS(X)

62c

3-11

X.NEQ Y

27c

3-3, 3-12

NIL

VAR

3-2, 3-7

.NOT Y

LAMBDA

3-12

ONES(X)

85c

3-11

αOP F αIS X

10c

3-1, 3-7

OPVARS()

12c

3-17

OR(:X)

80c

3-11

X.OR Y

35c

3-12

OUTPUT(X)

46c

3-14

PAIR(X,Y)

LAMBDA

3-6

PASN(X,V)

03c

3-8

PAUSE()

LAMBDA

3-18

PREF(X)

02c

3-7

αPROG X

LAMBDA

3-10

PROP(X,Y)

LAMBDA

3-6

PWD(X,S,L)

83c

3-11

QT(X)

17c

2-2, 2-6, 3-9

QTL(:X)

29c

3-10

READSEXP()

48c

3-14

REALSPACE

SET10

3-18, 4-2

REF(X,Y)

00c

3-5

REMPROP(X,Y)

LAMBDA

3-6

X.RES Y

38c

3-11

REVERSE(X)

LAMBDA

3-4

RHO(X)

44c

3-5

X.RTP Y

57c

3-11

SAVE(WSID)

51c

3-16

αSCOPE X

07c

3-7

SEARCH(X,P,F,U)

LAMBDA

3-4

SELECT(Q,:X)

LAMBDA

3-10

SEQ(:X,Y)

40c

3-9

SET(X)

55c

3-18

SET(X,V)

55c

3-15, 3-17, 4-2

SIGN(X)

63c

3-11

SIN(X)

68c

3-11

SINH(X)

70c

3-11

<u>Symbol and Form</u>	<u>Type</u>	<u>Page</u>
SLOAD(NAME)	52c	3-17
SPLIMINC	SET5	3-18, 4-1, 4-2
SQRT(X)	78c	3-11
STACKSPACE	SET11	3-18, 4-2
SUBLIS(X,Y)	LAMBDA	3-4
SUBSCRIPT(Y,:X)	04c	3-8
SUBST(X,Y,Z)	LAMBDA	3-4
SYMBOL()	56c	3-4
SYMBOL(X)	56c	3-4
TAN(X)	66c	3-11
TANH(X)	72c	3-11
TIMEOFDAY	SET14	3-18
TRACE	SET6	3-17, 3-18
αTRACE X	LAMBDA	3-18
TRANSLATE(X)	21c	3-17
TREE	SET7	3-17, 3-18
TRUE	VAR	3-2
TTY	VAR	5-1
UNBIND(X)	32c	3-14
UNTRACE(X)	LAMBDA	3-18
UQT(X)	18c	2-6, 3-9
USER	VAR	3-16
αWHILE C αDO X	19c	3-9
WIDTH	SET2	3-15, 3-18
WSID	VAR	3-16
XOR(:X)	82c	3-11
[:X;Y]	40c	2-3, 3-9
Y[:X]	04c	2-4, 3-8
Y[:X,Z]+V	05c	2-5, 3-8
X←V	06c	2-5, 3-7
X:=V	06c	2-5, 3-7
X	--c	2-2, 3-7
X=Y	26c	3-2, 3-12
X≠Y	27c	3-3, 3-12
X<Y	LAMBDA	3-12
X>Y	28c	3-12
X+Y	35c	3-11
X-Y	36c	3-11
-X	36c	3-11
X×Y	39c	3-11
X/Y	54c	3-11
X÷Y	37c	3-11
X Y	38c	3-11
%	VAR	A-1
α	VAR	A-1
'X'	17c	2-3
''	VAR	2-3
:X	VAR	3-1
αα	VAR	3-4

one of the following

1941-1942

CAMERO

1871

STACSPACE

17. 8. 1942

7. 1915

1. 8. 72 52
1. 10. 72

(continued)

121012

(X) 31.11.11

LYMAN

EXHIBIT

20452

X 30.12.19

1911-12-13

301

30.11
1950

100

1950

12-15-76

EXTRA

115-311

0-0 22744

1114

65

(X:) 153X

194

[Y. 13]

1875-1876

8-2-23, N. J. L.

41

...

$$y = y$$

458

427

248

1 + 1

2-8

Page 10 of 10

1938

1907

APPENDIX C

TABLE OF LISP 1.5 CORRESPONDENCES

This table gives the B 6700/LISP symbols that would be used in situations similar to many of the standard LISP 1.5 symbols. In most cases the symbols name functions. The table is ordered alphabetically on the LISP 1.5 symbols.

AND	.AND	GENSYM	SYMBOL
APPEND	APPEND	GET	REF
APPLY	APPLY	GO	αGOTO
ATOM	ISATOM	GREATERP	.GTR
CAR	CAR	IF	αIF αTHEN αELSE
CDR	CDR	INTERN	SYMBOL
CONC	CONC	LABEL	αOP αIS
COND	αIF αTHEN αELSE	LENGTH	RHO
CONS	CONS	LESSP	.LSS
COPY	COPY	LIST	LIST
CSET	PASN	LOGAND	AND
CSETQ	:=	LOGOR	OR
DIFFERENCE	-	LOGXOR	XOR
DIVIDE	÷	MAX	MAX
EFFACE	EFFACE	MEMBER	ISMEMBER
EQ	=	MIN	MIN
EQUAL	ISSAME	MINUS	-
EVAL	EVAL	NCONC	CONC
EVALQUOTE	APPLY	NIL	NIL
EXPT	RTP	NOT	.NOT
F	FALSE	NUMBERP	ISNUM
FUNCTION	QT	OR	.OR

READ	INPUT
RECLAIM	GCL
REMAINDER	.RES
REMPROP	REMPROP
RETURN	FIRST
REVERSE	REVERSE
RPLACA	ASN
RPLACD	ASNCDR

UNTRACE

UNTRACE

AND	AND
APPROX	APPROX
AREA	AREA
AVG	AVG
CAN	CAN
CUR	CUR
COMB	COMB
ALT	ALT
COVS	COVS
COY	COY
CSFT	CSFT
CSST	CSST
DIFFERENCE	DIFFERENCE
DIVIDE	DIVIDE
EFFECT	EFFECT
EQ	EQ
EXACT	EXACT
EXAM	EXAM
EXPL	EXPL
INT	INT
INSTR	INSTR

APPENDIX D

A COMPLETE LISP PROGRAM:

EXTENDING THE OPERATORS

FOR

COMBINATORIAL ANALYSIS

$$P(N, S, K) = \binom{N}{K} \binom{N-K}{S-K} = \binom{N}{S} \binom{N-S}{K-S}$$

$$N \geq S, S \geq K \geq 1$$

where N, S , and K are all integers

$$CP Y = Y$$

$$X CP Y = \binom{X}{Y}$$

APPENDIX D

A COMPLETE LIST PROGRAM: EXTENDING THE OPERATORS FOR COMBINATORIAL ANALYSIS

The problem to be considered here is the use of LISP for a parametric study of the following function of three variables:

$$P(N, S, K) = \binom{N}{S} \binom{S}{K} K! + \binom{N}{S-K} \binom{S-K}{K} K!$$

This function arises in the analysis of the complexity of sequencing algorithms. The variables N , S , and K represent the number of objects being sequenced, the cardinality of the subset being considered, and the level interaction between objects, respectively. The variables are constrained by

$$S \leq N$$

$$2 \cdot K \leq S$$

$$N \geq 2, S \geq 2, K \geq 1$$

where N , S , and K are all integers

In a parametric study of the values of $P(N, S, K)$, it is first recognized that LISP does not include appropriate operators for the binomial coefficients, $\binom{X}{Y}$, or the factorials, $Y!$. As a first step in the analysis, the operator set will be extended to include the infix operator, $.CP$, which is the acronym for combinations or permutations. This new operator will have both monadic and diadic forms defined by

$$.CP Y \equiv Y!$$

$$X.CP Y \equiv \binom{X}{Y}$$

The program to implement this operator would be the following:

```

aOP (:X) .CP Y aIS [ aIF Y=0 aTHEN 1
aELSE aIF X=NIL aTHEN Y*(.CP Y-1)
aELSE aIF (X+X[0])=Y aTHEN 1
aELSE ((X-1) .CP Y-1)+(X-1) .CP Y]
GCL()
[USER+QT(NEWPERSON); SAVE(QT(CPFILE))]

```

At this point the file called LISP/NEWPERSON/CPFILE is the original workspace extended to include the operator, .CP. The input/output record for this program would appear as follows:

LISP
CLEAR

```

aOP (:X) .CP Y aIS [ aIF Y=0 aTHEN 1
aELSE aIF X=NIL aTHEN Y*(.CP Y-1)
aELSE aIF (X+X[0])=Y aTHEN 1
aELSE ((X-1) .CP Y-1)+(X-1) .CP Y]

```

.CP

GCL()

231

[USER+QT(NEWPERSON); SAVE(QT(CPFILE))]

CPFILE

In the parametric study of the function, $P(N, S, K)$, the function will first be defined as a functional form operator, P , and evaluations of the operator, P , will be imbedded within an iteration over the desired domain of the variables. The complete input/output record for this part of the program would appear as follows:

```

USER+QT(NEWPERSON)
NEWPERSON
LOAD(QT(CPFILE))
CPFILE
  αOP P(N,S,K) αIS
    (((N .CP S)×(S .CP K))
    +(N .CP S-K)×((S-K).CP K))×(.CP K)
P
  [N+2; αWHILE N .NEQ 9 αDO [
    S+2; αWHILE S .NEQ N+1 αDO [
    K+1; αWHILE .NOT (2×K)>S αDO [
      OUTPUT(CONS(N,S,K,P(N,S,K),NIL));
      K+K+1]; S+S+1]; N+N+1]]
( 2 2 1 4)
(.3 2 1 9)
( 3 3 1 9)
( 4 2 1 16)
( 4 3 1 24)
( 4 4 1 16)
( 4 4 2 24)

( 8 8 4 3360)
END()
)END OF FILE

```

This example has been constructed to illustrate operator extensions and a typical use of LISP for numerical computations. Recursion was used freely in the programs in order to simplify the example, but more efficient formulations would eliminate much of the recursion. Such alterations in the algorithms in the interest of efficiency are easily implemented in a conversational system such as B 6700/LISP.