INTERPRETATIVE SYSTEM FOR THE PROGRAMMING OF RECURSIVE FUNCTIONS ON A DIGITAL COMPUTER

by

Jan G Kent

CONTENTS

PART II LISP IMPLEMENTATION

1 INTRODUCTION

This paper describes the implementating of LISP 1.5 on CDC 3600 at Kjeller Computer Installation.

The CDC 3600 is a 1's complement binary computer with a 48-bit word length and 32768 words of storage. Core speed is 1.5 microseconds. SCOPE, the monitor for CDC 3600 occupies 6000_{10} words of core store.

The machine has an accumulator A, an accumulator extension Ω and a flag register D. A, Ω and D are all 48 bits long. In addition there are six 15-bit index-registers B1 - B6 and various other registers.

LISP3600 as this interpreter is called has been modelled after the original LISP 1.5 (1) for the IBM 7090, and care has been taken to ensure compatibility between these two version.

The actual implementation of this interpreter differs in some important respects from the original version to increase the efficiency.

The most marked differences are in the organization of storage, where the idea of a separate block for "fullwordstorage" has been abandoned, and in the propertylist of each LISP-atom where the indicator PNAME never is needed. Note that the interpreter and the initial object list is assembled relocatable.

2 ORGANIZATION OF THE SYSTEM

2.1 Organization of storage

Core store is distributed according to this figure.

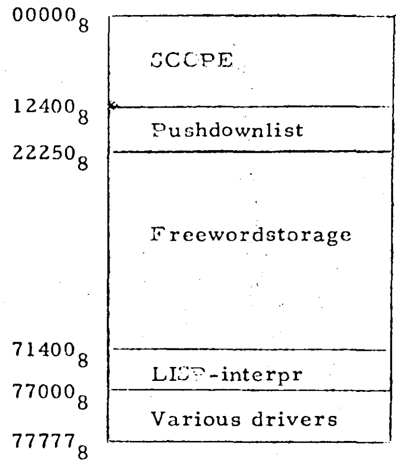


Figure 2.1 Organization of core store

The boundary between freewordstorage and the pushdownlist is fixed, though easy to reset when reassembling the LISP-system.

The length of the pushdownlist has been set to 4000_{10} words. The interpreter occupies about 2850_{10} words and freewordstorage the rest of core store, about $20\,000_{10}$ words. This compares favourably with LISP 1.5 which by excising LAP and the compiler has $16\,300_{10}$ words of free- and fullwordstorage and 2560_{10} words of pushdownlist.

2.2 Arguments and registers

Between LISP-functions arguments are transmitted through the A-register, Ω-register and the standard cells ARG3,.... .

The A-register is also used for transmitting addresses to and from the pushdownlist.

The value of a function is always held in A when returning. The D-register is used to hold several indicators (binary switches) needed in the interpreter. Information about the status of the interpreter can then be read out of the bit for bit displayed D-register on the console. The reader is referred to Appendix III for a description of the indicators.

2.3 Freewordlist

All unused words in freewordstorage are strung together on a list called the freewordlist. Every time a new word is needed it is taken from this list. The address of the first word on the freewordlist is always held in a location called FREE.

2.4 Object list

In LISP all predefined atoms are held on a list called the object list. That part of the object list which contains the standard atoms has been generated in assembly language. See Appendix VI. When an atom is encountered by the reader this list is searched to see if the atom already exists, if not the atom is appended to the object list. The address of the first word on the object list is held in the location OBJECT, which is accessible to the LISP-programmer as the property APVAL of the atom OBLIST.

2.5 Property list

Some atoms have special properties. Information about atoms is stored in the atom's property list. A typical property list might look like this:

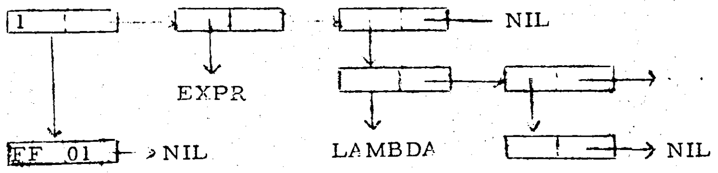


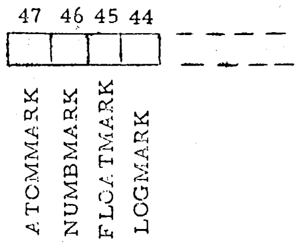
Figure 2.2 Property list of the atom with printname FF

FF is as we see a function namely an EXPR which starts this way: (LAMBDA (X) .....).

2.5.1 Binary markers

As the wordlength is 48 bits, and only 15 bits are needed to express an address, 9 bits in the upper halfword and 9 bits in the lower halfword are released for other uses.

If bit 47 in a word is set this indicates that the word is an atomhead.



47	46	45	44	
1	0	0	0	BCD ALFAMER
1	1	0	0	FIXPOINT
1	1	1	0	FLOAT POINT
1	1	0	1	LOGICAL

Figure 2.3 Markers and their meaning in the atomhead

The bits 46, 45 and 44 refer to the word whose address is in the upper address of the atomhead, (see Figures 2.2 and 2.3). This word is the fullword list associated with the atom. Bit 39 indicates whether or not a function is to be traced.

When bit 22 is set this indicates that the word in question is a fullword. Bit 23 is used by the garbage collector to mark active words.

2.5.2 Fullwords

The fullwords in freewordstorage replace the "fullwordstorage" in LISP 1.5.

A fullword is a word with the 24 upper bits occupied by either:

a) Four BCD characters from a printname. (If need be, filled in from the right with blanks.)

or

b) 24 bits from a 48-bit number.

or

c) The address of a binary LISP-routine (SUBR or FSUBR).

2.5.3 Printnames

All nonnumeric atoms have in the upper half of their atomhead the address of a linear list of their BCD printnames. For instance the atom DIFFERENCE had this fullwordlist:

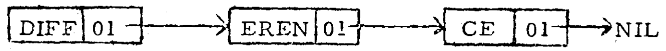


Figure 2.4 The fullwordlist of the atom DIFFERENCE

2.5.4 Numbers

There are three kinds of numbers:

a) Fixed point

b) Floating point

c) Logical

All are stored as 48-bit binary numbers with the help of two fullwords, and must be converted from or to BCD in input and output. (The BCD representation of a number is not stored.)

2.6 Organization of input and output

Input and output operations in the interpreter always refer to so-called logical units. The concept of logical units is introduced by SCOPE to help all programmers achieve flexibility in input and output. A logical unit is independent of actual units in a given machine-configuration. The LISP-programmer chooses his actual input and output units at run-time. All output in the interpreter goes to logical unit 11 and all input is from logical unit 10.

3 THE INTERPRETER

The following chapter is a description of the main subroutines in the interpreter which cannot be described in LISP. Note that flow charts for some of these routines are given in Appendix I. In this description all words written with capital letters (except the register mnemonics) refer to symbolic addresses in the interpreter, or to the names of indicators in the D-register. (See Appendix II for a listing of the interpreter and Appendix III for a description of the indicators.) All functions which can be defined in LISP are given in M-expressions in the manual LISP3600: Users Manual.

3.1 Recursion techniques

LISP is a very recursive programming system as can be seen from the fact that a LISP function may call the LISP-interpreter which then in effect has to interpret a call to itself. Allimportant in all this recursion is the pushdownlist and the bookkeeping (SAVE and UNSAVE) of the pushdownlist.

3.1.1 The pushdownlist (Stack)

The whole system uses only one pushdownlist. This pushdownlist then has to hold:

a) Arguments of recursive routines. (Always in the form of pointers to lists in freewordstorage.)

b) Return address for routines entered by a recursive call.

c) Addresses of specific operations to be performed in combined routines, when these have to call the interpreter to evaluate their arguments (see under arithmetic routines).

Note that the pushdownlist is just a linear block of storage with no list-structure.

3.1.2 SAVE

This subroutine puts the A-register on top of the pushdownlist and increases the pointer which points to the top. If the stack is exceeded control is transferred to ERROR.

3.1.3 UNSAVE

This subroutine decreases the pointer to the top of the pushdownlist, and loads A with the word on top. If the bottom of the pushdownlist is reached control is transferred to ERROR.

The recursive call on, and return from, a routine is handled by two subroutines named CALL and RETURN. Within the interpreter the calling sequence needed for calling a routine by the help of CALL is inserted by a macro called RGJPCALL.

3.1.4 RGJPCALL

This macro has a single argument namely the name (symbolic address) of the routine to be called. The macro also preserves index register B1. This has proven very useful because it means that B1 can be used freely in all routines entered by CALL.

RGJPCALL retains control if the argument is CAR or CDR and codes in the necessary 5 instructions which perform CAR or CDR.

3.1.5 CALL

This subroutine is entered by a return jump (from RGJPCALL usually) which means that a jump back to the calling routine is stored in the very first location in CALL. This location is then saved by the help of SAVE. The A-register is preserved in this operation. Lastly control is transferred to the routine which was to be called.

3.1.6 RETURN

This subroutine unsaves the jump mentioned under 3.1.5 and executes it, thereby returning control to the calling routine.

3.2 The reader

All symbolic analysis is done in LISP by the reader, which is usually written first. The reader consists of the routine READ which uses GETCHAR, TRYATOM, TRYRPAR, TRYDOT and STORATOM.

When control is transferred to READ a single S-expression is read from logical unit 10. The address of the first word in the internally generated list is held in A when returning from READ.

3.2.1 READ SUBR pseudofunction

In the flow chart for READ some LISP-terms are used to explain the working of the reader. However, no LISP-functions are actually used in READ, mainly because the reader was written first.

READ is a recursive subroutine. The recursion is performed by a subset of READ with two entry points UPPER and LOWER. The subset is called recursively each time a leftparenthesis is encountered in the input string. If the leftparenthesis is the first character after a dot the entry point LOWER is used, otherwise the entry point UPPER is used.

The statement "Try to read atom into car [(A)]" means that TRYATOM is called which, as the name implies, seeks to make an atom out of the next characters in the input string.

Control is returned to READ if a nonalphameric character (except preceding blanks) is encountered. The same holds for the other "Try to" except that these routines only read a single character (though skipping preceding blanks) and exits successfully or unsuccessfully.

READ also checks for syntactical errors and puts in an errortext at the point where the error was encountered. At the same time the error-indicator ERRIND is set. This indicator is checked before entering EVALQUOTE and if it is on EVALQUOTE is skipped and an error-printout effected.

3.2.2 GETCHAR

This subroutine loads A with the next character from the input buffer BUFF (0-10). As the input is always cards or cardimages a new card is read each time 72 characters have been read. The number of characters read within a card is held in TEMP+2. If the character in A is an apostrophe, the running of the interpreter is stopped and control returned to the monitor SCOPE. (That is to say, apostrophe acts like "end of file"-mark to LISP3600).

3.2.3 TRYATOM

This subroutine tries to form a string of BCD characters or a half converted number in BUFF(11-20). If the next characters in the input string makes this possible. TRYATOM calls STORATOM which makes atoms out of the information in BUFF(11-20). To aid TRYATOM in the understanding of the input string and STORATOM in the making of atoms several indicators in the D-register is used by TRYATOM. All these indicators are cleared when entering TRYATOM.

The possible indicators used by TRYATOM are:

Bit no	Name	Usage
10	ATOMIND	This is set if the first character (skipping preceding blanks) is alphamerical, and is used to tell TRYATOM that the construction of an atom is in progress.
11	NUMBIND	Set if first character is a digit. Used to tell TRYATOM that the construction of a number is in progress, and to tell STORATOM that a numeric atom must be made.
12	MINUSIND	Set if first character is a minus. Tells STORATOM that the number must be complemented before atommaking.
13	FLOATIND	This is set if a decimal point is encountered under the construction of a number. Tells TRYATOM that the number encountered is floating point, and STORATOM that floating-point conversion must be done.
14	EKSPIND	Set if the letter E is encountered while assembling a floating-point number. Signals TRYATOM and STORATOM that the number has an exponent.
15	LOGIND	Set if the letter Q is encountered while assembling a number. Signals TRYATOM and STORATOM that this is an octal number with a possible scale factor after the Q.
16	NEGEKSIND	Set if a minus is encountered while the EKSPIND is on. Signals that the exponent is negative.
17	LETTIND	Set if first character is a letter. Indicates that the atom is nonnumeric.

Note that the characters are brought by GETCHAR, and the last brought is always held in LASTCHR. The first non-blank character encountered by TRYATOM in the input string determines if it is possible to make an atom or not, namely if the character is alphameric or not. If atom-making is possible the mode of conversion is also set by the first character. A special mode is set if the first two characters are $$. The character following $$ is preserved (called DELIMITR) and all characters after this put into the conversion area without checking until DELIMITR is encountered again. This provides for making atoms containing arbitrary characters.

Exits from TRYATOM are also governed by the first non-blank character encountered as can be seen from the following table.

Address of the instruction calling TRYATOM is called CALLING.

Condition	Returnaddress
Successful atommaking	CALLING + 1
Comma	CALLING + 2
Dot and right parenthesis	CALLING + 3
Left parenthesis	CALLING + 4

If an atom has been made its address is in Q when returning to READ.

3.2.4 STORATOM

This subroutine converts and makes an atom out of information in the conversion area BUFF(11-20) according to the indicators in the D-register set by TRYATOM. These indicators are also used to set the correct markers in the atom's head. When the construction is finished the entire object list is scanned to see if this atom has been generated before. If the atom already exists the address of the elder atom is in A when returning. If the atom is new it is appended to the object list and its address held in A when returning.

3.2.5 TRYRPAR

This subroutine compares the first character (skipping preceding blanks) with a right parenthesis, and if they are equal returns to CALLING + 1, otherwise return is to CALLING + 2. (CALLING is as before the address of the instruction calling the subroutine in question.) Characters are brought by GETCHAR, and the last brought held in LASTCHR.

3.2.6 TRYDOT

This subroutine checks for a dot, in the same way as TRYRPAR.

3.3 The printer

Almost all output of S-expressions is handled by the printer. The printer consists of the SUBR PRINT which uses the subroutine SENDATOM and WRLINE.

3.3.1 PRINT SUBR pseudofunction

If the argument of PRINT is nonatomic A is set to minus zero and saved, thereafter PRINT starts an iteration the end of which is signalled by an unsaving of minus zero. In this iteration the list-structure which is the argument of PRINT is converted to S-expressions and placed in the output buffer OUTBUFF(0-15) for later printing by WRLINE. As the S-expressions may be longer than 15 words there is a check for end of line. If the word OUTBUFF + 10 is filled, bit 3 in the D-register (OVERIND) is set, the line terminated and printed before a new atom or a new sublist is built up. The conversion to S-expressions then continues from OUTBUFF + 1 again.

3.3.2 SENDATOM

This subroutine sends a printname to the output buffer after performing the necessary conversions. The arguments are the atomhead of the atom in question held in Q and a pointer to its fullwordlist in B2. SENDATOM uses bit 15 in the D-register (LOGIND) to indicate that a logical number is under conversion. The subroutines for converting internal binary representation of logical, fixed point and floating point numbers to external BCD representation are all standard Kjeller Computer Installation routines. The routine for output of floating point numbers gives in this version 1 digit before the decimal point and 6 after. Trailing zeros in the mantissa are omitted and the same holds for leading zeros in the exponent.

3.3.3 WRLINE

This subroutine transmits a line to logical unit 11. Argument is the address (in A) of the control word to be used. If this is the standard control word for printing from OUTBUFF, the words containing the line (OUTBUFF (0-15)) is transmitted to OUTBUFF (20-35) and printing initiated from OUTBUFF + 20. At the same time OUTBUFF (0-15) is reset to blanks.

If the control word address is the address of any other control word, WRLINE only initiates printing and returns control.

3.4 Other input and output routines

READ will read a BCD list from logical unit 10, PRINT will write an internal list-structure on logical unit 11. In order to process non-list input and output, LISP has several generalized input and output routines. When using these routines together with the routines which operates on the printnames, extreme generality can be achieved in input and output.

3.4.1 ADVANCE SUBR pseudofunction

ADVANCE reads the next character from the card currently in the input buffer and returns with it made into an atom. This is done by first calling GERCHAR and then sending the character to BUFF + 11, after which STORATOM is called. ADVANCE checks TEMP + 2 to find out how many characters have been read. If the 72 characters have been read ADVANCE increases TEMP + 2 and returns with the atom EOR as value. After reading EOR, the next ADVANCE will bring the first character on the next card by calling GETCHAR etc.

3.4.2 STARTREAD SUBR pseudofunction

STARTREAD always brings the first character on the next card. This is done by setting TEMP + 2 to 80 and jumping to ADVANCE.

3.4.3 PRIN1 SUBR pseudofunction

PRIN1 has an atom as argument which it sends to the output buffer. This is done by calling SENDATOM. As SENDATOM has checking for

end of line, executing successive PRIN1's will fill up and print out line after line.

3.4.4 TERPRI SUBR pseudofunction

TERPRI prints out the output buffer by calling WRLINE.

3.5 Routines operating on printnames

3.5.1 PACK SUBR pseudofunction

The argument of PACK should be an atom, and the effect of PACK is to place the first character in this atom's printname in the output-buffer.

3.5.2 MKATOM SUBR pseudofunction

MKATOM makes an atom out of the characters placed in the output-buffer by PACK. This is done by calling STORATOM. The value of MKATOM is the newly created atom (see STORATOM for details). Executing MKATOM without first having executed PACK gives the atom BLANK. MKATOM is a new function in LISP3600, and in terms of LISP 1.5 its effect is: MKATOM ≡ INTERN (MKNAM).

3.5.3 UNPACK SUBR pseudofunction

The argument of UNPACK should be an atom. The value of UNPACK is a string of atoms each having as printname a single character from the original atom. This is done by calling SENDATOM and then transmitting the characters in the printname one by one to BUFF + 11 while calling STORATOM.

3.6 Elementary functions

3.6.1 CAR SUBR

CAR loads A with the upper address of the word, whose address was the argument of CAR.

3.6.2 CDR SUBR

CDR loads A with the upper address of the word, whose address was the argument of CDR.

3.6.3 CONS SUBR

a) CONS obtains a new word from the freewordlist by taking the first word on the freewordlist.

b) If this word is not the last word on the freewordlist FREE is set to CDR of the freewordlist.

c) Thereafter CONS places its two arguments in upper and lower address of the new word, and returns with the address of this word as a value.

d) If the new word was the last on the freewordlist, the garbage collector is called. Upon return to CONS FREE is checked for zero and if zero, ERROR is called with the remark "store is full". If nonzero the first word is again tested for being the last and if not, CONS proceeds as described under a), b) and c).

If the freewordlist again contained only a single word ERROR is called with the same remark as under d).

Note: Wherever an unused word is needed from freewordstorage in the interpreter CONS is used to bring it. This also applies to fullwords, see 2. 5. 2.

This also means that the garbage collector can only be called from CONS.

3.6.4 ATOM SUBR predicate

Bit 47 in the word, whose address was the argument of ATOM is checked. If zero the value of ATOM is F, if nonzero the value is T, see 2. 5. 1.

3.6.5 EQ SUBR predicate

If the two addresses which are the arguments of EQ are equal, the value of EQ is T, otherwise F.

3.7 The garbage collector

The garbage collector is called from CONS whenever the freewordlist has been exhausted, and unless the computation is too large for the system, there are many words in freewordstorage that are no longer needed. The garbage collector uses these to make a new freewordlist.

To find the unused words, the garbage collector sets bit 23 in all needed words.

Since it is important that all needed lists be marked, the garbage collector starts marking from several base positions:

a) The object list, whose starting address is in the location OBJECT. This protects the atomic symbols, and all list-structures that hang on the propertylists of atomic symbols.

b) The portion of the pushdownlist that is currently being used. This protects most of the results of the computation that is in progress.

c) The two arguments for CONS, when the garbage collector was called.

d) The location READTEMP + 3 which hold the starting address of either the list-structure under construction by READ, or the last read list-structure.

e) The locations TEMPORAR (0-5) which holds intermediate results in computations involving CONS.

f) The locations ALIST and ARG3 (0-20) which holds arguments for the interpreter or LISP-functions.

Before using any of the addresses in the base positions, they are tested for being pointing into freewordstorage. If they are pointing into freewordstorage RECRGARB is called, if not the next base position is tested and so on. When all base positions have been run through the garbage collector scans freewordstorage linearly, setting bit 23 to zero in all marked words, and stringing all unmarked words together into a new freewordlist, whose starting address is put into FREE before returning. If the end of freewordstorage is reached without discovering a single unmarked word, FREE is set to zero before returning.

The garbage collector is also used to initialize freewordstorage. Before entering the garbage collector for this initializing, bit 46 in the D-register is set. This bit tells the garbage collector that it need only mark the object list.

3.7.1 RECRGARB

This recursive subroutine is used by the garbage collector to perform the actual marking of a needed list, whose starting address is in B1 when entering.

Marking proceeds as follows:

First every needed word within freewordstorage that can be reached through a CAR-CDR chain from a base position is marked by setting bit 23. Whenever a word with bit 23 set is reached in a chain during this process, the garbage collector knows that the rest of the list involving this word has already been marked, and does not mark again.

A CDR chain is stopped by either NIL or zero in the lower address of a word.

A CAR chain is stopped by either zero in the upper address of a word or by reaching a fullword (bit 22 set). The marking of a fullwordlist is done by a special part of RECRGARB which never takes CAR of a word. This is necessary because the upper address of a fullword must never be used as a pointer (see 2.5.2). The testing for zero is necessary because the garbage collector can be called from routines such as READ which constructs lists, and all pointers in a list-structure may not be known at the time. The fact that the address in a word will be put in later is signalled by setting the address to zero.

A garbage collection can be recognised by bit 47 in the D-register. This bit is set upon entering the garbage collector, and cleared before returning control to CONS.

3.8 The interpreting routines

Some of the functions that can be defined in LISP are given in M-expressions in Appendix II.

The LISP definition of these functions have been followed with one exception, EVLIS. To make the routines faster all unneeded recursion as indicated by the LISP definitions has been changed to iteration. This left only four places where recursion was needed, one place in APPLY and three in EVAL. In these places recursion were needed because of having to call EVAL or EVLIS before applying APPLY.

To eliminate recursion in EVLIS a new routine APPEND1 was written. This routine has been very useful and is used in PROG, DEFINE and APPEND apart from in EVLIS.

EVLIS and LIST are in effect the same binary program.

The routine ERROR is always called directly in the interpreter, and a special section of ERROR takes care of these calls.

3.8.1 ERROR SUBR

The function ERROR will cause an error diagnostic to occur. The argument (if any) of ERROR will be printed. ERROR is of some use as a debugging aid.

As mentioned above a special section of ERROR gets control if an error occurs in the interpreter. Before transferring control to this section A and Q must be loaded with a specification of the error such as "A7 SPREAD". When entering the special section A and Q are stored in a standard error diagnostic line and printed out. Thereafter all lists bound on the pushdownlist are printed out.

ERROR always stops interpreting and gives control back to the main program, to read the next doublet. See LISP3600: Users Manual for a complete listing of all error diagnostics.

3.8.2 SPREAD

This subroutine is not available to the LISP-programmer. SPREAD can be regarded as a pseudofunction of one argument. The argument is a list. SPREAD puts the individual items of this list into the standard cells A, Q, ARG3, ... for transmitting arguments to functions.

3.9 The trace feature

This feature has been implemented in an unconventional way in LISP3600. Though externally operating in much the same way as in LISP 1.5, internally the difference is marked. At present only EXPRs can be traced, and the EXPRs are only checked for tracing in EVAL.

The checking is only performed if the TRACEIND (bit 7 in the D-register) is on. If this indicator is off the checking is skipped and all EXPRs are evaluated faster.

The TRACEIND can be set by executing SETBIT (7) and cleared by executing CLEARBIT (7).

Tracing is also controlled by the pseudofunctions TRACE and UNTRACE:

If an EXPR has been the argument of TRACE and the TRACEIND is on, the name and arguments of this EXPR will be printed when it is entered recursively and its name and value when it is finished. Thus there are in EVAL two checks for tracing of an EXPR. The first is after evaluating its arguments, to see if they should be printed, and the other is when returning with its value from APPLY, to see if this should be printed. Arguments are printed out via the subroutine TRACEARG and value via the subroutine TRACEOUT. Both subroutines also print the name of the EXPR. The check for tracing is made possible by saving a pointer to the EXPR in question while evaluating its arguments and while finding its value. This pointer is also utilized by TRACEARG and TRACEOUT when they print the EXPRs name.

3.9.1 TRACE SUBR

The argument of TRACE is a list of functions to be traced. TRACE set bit 39 in the atomheads of all of them and this bit is then checked in EVAL if the TRACEIND is on.

3.9.2 UNTRACE SUBR

This function removes tracing from all the functions in the list, which is the argument of UNTRACE. This is done by clearing bit 39 in the atomheads of these functions.

3.10 Arithmetic functions

The arithmetic functions in LISP3600 needs special mentioning. To save space all functions which are equal in all respects save the actual operation involved are combined into a single routine with different entrypoints. For instance the function PLUS and TIMES uses the same routine called PLUSTIME, with the two entrypoints PLUS and TIMES. PLUSTIME (as the other combined routines) performs the correct operation by executing it indirectly through the location ADR., which is loaded with the address of the correct instruction at the entry-point. In other words the address of an addinstruction is placed in ADR. at the entrypoint PLUS prior to transferring control to PLUS-TIME. If the functions to be combined into single routines are FSUBR, the address in ADR. must be saved before evaluating an argument and unsaved afterwards. The reason is obvious; for instance PLUS may very well have TIMES as an argument, and as they both use PLUS-TIME something must be done to preserve the operation while evaluating arguments.

For the same reason intermediate results are saved, and unsaved while evaluating arguments.

Functions with numbers as values transfer control to the routine STORE, when they have finished the computation. The result must be placed in PLUS1 + 1 and some indicators set prior to jumping to STORE.

STORE then generates a non-unique numberatom with the binary markers in the atomhead set according to the indicators in the D-register.

3.10.1 STORE

This subroutine generates a non-unique number.

If the FLOATIND is on, the FLOAT marker is set in the number's atomhead.

If the LOGIND is on, the LOG marker is set in the number's atomhead.

If none of these are on a fixpoint number will always be generated.

3.10.2 Arithmetic functions in LISP3600

Function	Type	Routine used	Numb of args	Result	Remarks
PLUS	FSUBR	PLUSTIME	indefinite	$x_1+x_2+..+x_n$	
TIMES	FSUBR	PLUSTIME	indefinite	$x_1 \cdot x_2 \cdot .. \cdot x_n$	
DIFFERENCE	SUBR	DIFFQUCT	2	x_1-x_2	
QUOTIENT	SUBR	DIFFQUCT	2	x_1/x_2	
ADD1	SUBR	ADD1SUB1	1	x_1+1	
SUB1	SUBR	ADD1SUB1	1	x_1-1	
EXPT	SUBR	EXPT	2	$x_1^{x_2}$	Result is always floating point
LEFTSHIFT	SUBR	LEFTSHIF	2	$x_1 \cdot 2^{x_2}$	
REMAINDER	SUBR	REMAINDE	2	Remainder of x_1/x_2	Takes only fixed-point args
MINUS	SUBR	MINUS	1	$-x_1$	
LESSP	SUBR, pred	GT. LT.	2	$x_1 < x_2$	
GREATERP	SUBR, pred	GT. LT.	2	$x_1 > x_2$	
ZEROP	SUBR, pred	ZEROP	1	$x_1 = 0$	
MINUSP	SUBR, pred	MINUSP	1	$x_1 < 0$	
EQUAL	SUBR, pred	NUMBEQ	2	$x_1 = x_2$	
FIXP	SUBR, pred	FIXP	1	x is a fix-point-number	

Function	Type	Routine used	Numb of args	Result	Remarks
FLOATP	SUBR, pred	FLOATP	1	x is a float-point-number	
LOGP	SUBR, pred	LOGP	1	x is a logical number	
LOGAND	FSUBR	LOGFUNC	indefinite	$x_1 \wedge x_2 \wedge .. \wedge x_n$	
LOGOR	FSUBR	LOGFUNC	indefinite	$x_1 \vee x_2 \vee .. \vee x_n$	
LOGXOR	FSUBR	LOGFUNC	indefinite	$x_1 \vee x_2 \vee .. \vee x_n$	

3.11 The PROG feature

The PROG feature gets control when EVAL discovers the atom PROG as the first element of a form.

a) As soon as PROG is entered the PROGIND (bit 4 in the D-register) is set and the list of its program variables is used to make a new list in which each one is paired with NIL. This list is then put on top of the current association list. (The association list is a sort of working stack used by the interpreter to find the values of variables.) Thus all program variables have the value NIL at the entrance to the program.

b) The remainder of the program is searched for atomic symbols that are understood to be labels. A go-list is formed in which each label is paired with a pointer into the remainder of the program.

c) Then the execution of the program proper is started. If we skip the two first elements (PROG and the list of program variables) from the program, it can be regarded as a list of statements. The statements may be preceeded by a label. Since labels are always atomic and statements are not, a test for atom is necessary to discern between them.

Before executing a statement by calling EVAL, the go-list, the association list and a pointer to the statement in question followed by the rest of the program, is saved.

All statements are executed by calling EVAL and ignoring the value. This also applies to the functions SET, SETQ, GO and RETURN, all of which can only be used in PROG.

The function COND is acting somewhat differently inside a PROG feature, i e, if COND runs out of clauses error diagnostic A3 will not occur. Instead the next statement is executed. However, as all statements are executed by calling EVAL, EVCON is used to evaluate the COND in both cases.

EVCON tests the PROGIND to see if it should call ERROR or simply return if there are no true clauses.

But the PROGIND is turned off when leaving a PROG, and EVCON may call upon a new PROG when evaluating an if-clause. This problem was solved by making EVCON save the D-register before evaluating an if-clause.

The use of EVCON to evaluate both types of COND introduces a new problem: if a conventional function (not using PROG) is used within a function using PROG, the PROGIND will be on while the conventional function is evaluated. This means that error diagnostic A3 will not occur in the conventional function either. This problem has not yet been solved.

3.11.1 SET SUBR

The first argument of SET should be the name of a variable. The second argument is the new value to be given to this variable.

SET locates the name of the variable on the association list, and replaces its old value with the new.

If SET cannot find the variable on the association list, control is transferred to ERROR.

3.11.2 SETQ FSUBR

SETQ evaluates its second argument (the value) and jumps to SET.

3.11.3 GO FSUBR

GO unsaves from the pushdownlist until the address in PROG which calls EVAL to evaluate statements is found. This address is set aside and the next three locations, containing statement under execution, association list and go-list respectively, unsaved. The GO locates its argument on the go-list and preserves the pointer it finds there. The pushdownlist is then built up again by saving the go-list, the association list, the preserved pointer (the new statement to be executed) and the abovementioned address. Lastly control is returned to EVAL with the preserved pointer in the A-register, and EVAL starts executing the corresponding statement. If the label occurring in GO cannot be found on the go-list, control is transferred to ERROR.

3.11.4 RETURN SUBR

RETURN as the GO unsaves from the pushdownlist until the address in PROG which calls EVAL to evaluate statements is found. When this address is found and four more locations have been unsaved, control is returned to EVAL with the argument of RETURN in the A-register. This will in fact return control to the function calling PROG, with the argument of RETURN as value.

3.12 Miscellaneous functions

3.12.1 SETBIT SUBR

SETBIT sets bit no x in the D-register, where x is the argument of SETBIT.

3.12.2 TESTBIT SUBR

TESTBIT has value T if bit no x in the D-register is on, and F if it is off. x is the argument of TESTBIT.

3.12.3 CLEARBIT SUBR

CLEARBIT clears bit no x in the D-register, where x is the argument of CLEARBIT.

SETBIT, TESTBIT and CLEARBIT should not use the bits 0-5 and 10-17 in the D-register as they are reserved for other uses. Their arguments should be fixpoint numbers.

3.12.4 GENSYM SUBR

GENSYM generates a unique atom each time it is called, and returns with this as its value.

The new atom is of the form Gxxxxxxx where xxxxxxx is a number in the range 0-9999999. GENSYM works by having a number in the specified range in the location SYMBOL. This number is initially zero. The number is then converted to decimal and prefixed a G. GENSYM then changes an instruction in STORE, and calls it. The modifying of STORE makes this routine generate the new symbol as an alphameric atom not attached to the object list. When control is returned to GENSYM the instruction in STORE is corrected and the number in SYMBOL increased with 1. Thereafter control is returned to the calling function.

4 STATUS OF THE SYSTEM

CDC 3600 has an instruction repertoire which is particularly well suited to list-processing, and this together with the new ideas introduced has increased the speed of the interpreter. The functions that are implemented as SUBR and FSUBRs are chosen so that all LSIP 1.5 functions can be defined in LISP3600. There are five major exceptions to this rule: ARRAY, ERRORSET, COUNT, UNCOUNT and SPEAK have not been implemented.

All LISP 1.5 programs not using these functions can be run in LISP3600, with only trivial changes. See LISP3600: Users Manual for particulars. Several LISP 1.5 programs have in fact been run on CDC 3600 with no changes. Some notes about these and other LISP-programs can be found in LISP3600: Users Manual

4.1 Possible extensions of the system

The unwritten functions mentioned above, should be implemented. A control card interpreter should be written. The control card should be optional, and might contain a time limit, and some controlwords, such as TRACE, SET and SETSET.

A routine should be written which would call ERROR when the prescribed time had elapsed.

The controlword TRACE should indicate that the TRACEIND should be set, and SET and SETSET would write out on tape the freewordstorage after computation.

A controlword for reading in such a tape made by SET should also be introduced.

There are provisions in LISP3600 for using the usual mathematical symbols +, -, /, * and ** meaning exponentiation. An interpreting function that could take arithmetic expressions in infix notation using these symbols might be written.

LISP3600 was written for a one-bank CDC 3600, and must be used as such on multibank versions of CDC 3600.

On a CDC 3600 with two or more banks, numbered commen (the pushdownlist) may be placed in its own bank, if SAVE and UNSAVE are slightly modified. By doing this it should be possible to increase freewordstorage to about $29\ 000_{10}$ words.

The main problems connected with the ideas mentioned above have all been solved in theory, but unfortunately there has been no time to try them out in practice.