TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of contract SD-97

LISP 1.5 Reference Manual for Q-32

S. L. Kameny

9 August 1965

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

The views, conclusions or recommendations expressed in this document do not necessarily reflect the official views or policies of agencies of the United States Government.

LISP 1.5 Reference Manual for Q-32

ABSTRACT

This document is a reference manual for the Q-32 LISP system in operation under the Time-Sharing System (TSS) on the AN/FSQ-32 computer. It describes the working of the LISP system, and contains descriptions of all currently available and installed functions, except for input-output and library functions given in TM-2337/102/00.

This document conforms to the current numbering on LISP 1.5 documents, and supersedes TM-2430/000/00, which was a draft.

ACKNOWLEDGMENT

In the writing of this document, the author has mixed new material with descriptions adapted from the LISP 1.5 User's Manual and from the book The LISP Programming Language: Its Operation and Applications. He also wishes to acknowledge contributions in subject matter and clarification by Clark Weissman of SDC, Robert A. Saunders and Dr. Paul Abrahams of Information International, Incorporated, and Prof. Dan Bobrow of Massachusetts Institute of Technology.

The Q-32 LISP system is based on a compiler written in LISP and compiled by itself on an IBM 7090, then merged into a machine coded section assembled in SCAMP on the Q-32. The compiler was written by R. A. Saunders of I.I.I., based upon the Hart Compiler for the M-460 computer, with assistance from T. P. Hart, D. Edwards and M. Levin of M.I.T., and Prof. J. McCarthy and S. Russell of Stanford University. Some system functions were written by C. Weissman and the author.

CONTENTS

List of Figures

List of Tables

1. INTRODUCTION

This manual is intended for those already familiar with the LISP programming language. It contains a description of the internal mechanics of the Q-32 LISP system and supersedes any previous description of the Q-32 LISP system. The reader is referred to Ref. 1* and to the LISP 1.5 Manual (Ref. 2) for additional descriptions of LISP 1.5 language and its use. Input-output and library functions of Q-32 LISP 1.5 Mod. 2.5 are given in TM-2337/102/00; for Mod. 2.6, they are described in TM-2337/111/00.

The beginning LISP user will find the Q-32 LISP Primer (Ref. 3) useful. Further information on the Q-32 Time-Sharing System can be found in Ref. 4.

The Q-32 is a 1's complement binary computer with a 48-bit word length and 65,536 words of storage. Core speed is about 2 microseconds, and some instructions overlap. It has an accumulator, an accumulator extension called the B-register, eight index registers, and various other electronic registers. Peripheral equipment includes 16 tape drives (729 IV), about 700,000 words of drum storage, a card reader, card punch, and a line printer, 6 display consoles, a RAND tablet, and 50 remote typewriter stations. A PDP-1 is used as a peripheral processor to service time-shared teletypes. When run under time-sharing, the lowest 16,384 registers are used by the Executive.

The external language is compatible with LISP 1.5. Some features are not implemented at present. Most programs that will run with 7090 LISP will run on Q-32 LISP without change.

From the user's point of view, the Q-32 LISP system is seen through a version of Evalquote, which reads a pair and executes it. As in 7090 LISP, the pair is a function and a list of arguments. If the function is an atom carrying a functional definition, that definition, in the form of compiled code, is applied to the arguments. If it is a functional expression, the expression is compiled and then executed.

Because the Q-32 LISP system is compiler-oriented rather than interpreter-oriented, the user should expect:

- Programs to run faster on the Q-32 than (uncompiled) on the 7090.
- To have to pay more attention to variable declarations than in an interpretive system, where free variable bindings are available automatically.
- To have less thorough error-checking by the LISP system. In particular, there is no check to see that the proper number of arguments is supplied to functions.

* The Q-32 LISP system in current use is Mod. 2.5, and has been considerably changed from Mod. 1.0 system described in Ref. 1.

2. USING THE TIME-SHARING SYSTEM FOR LISP

In this section, the steps necessary to use the Q-32 LISP are discussed. These steps include both the communications with the time-sharing Executive and the communication with the LISP system itself.

2.1 GENERAL PROCEDURE FOR MESSAGE INPUT

Messages are sent to the Time-Sharing System in either of two modes: Executive mode or object program mode. The Executive mode is used for communicating messages to the time-sharing Executive, and the object program mode is used for communicating messages to LISP. With a few exceptions, the system stays in one mode until it is specifically instructed to shift to the other. The two characters, the exclamation point (!) and the quotation mark (") are used as mode control characters. The exclamation point is used to go into Executive mode, and the quote mark to go into object program mode. Typing an exclamation point has two effects: it sets the input mode to the Executive mode, and, if it is not the first character of the line (mode control characters excluded), then it also causes the line to be ignored. In the latter case, the system sends back a carriage return and line feed. The quotation mark also cancels a line, but sets the input mode to object program mode. (If already in object program mode, the quotation mark simply cancels the line.)

A message (in either mode) is terminated by a carriage return. Until the carriage return is received, the message is not sent from the PDP-1 to the Q-32; thus, a message cancelled by a mode control character is never seen by the Q-32. It is possible to cancel single characters as well as entire lines by using the "rub out" key on Model 33 and 35 TTY's. The effect of this key is to cause the last character to be ignored. It can be used several times in succession; for example, three rub outs in succession will cause the last three characters to be deleted from the input message. However, the effect of a rub out cannot be seen by examining the typed text.

At present, LISP acknowledges a need for input by ringing the bell. It is important to wait after typing a line before typing the next line when communicating with LISP. (If a line has been cancelled with a quotation mark, however, then the bell signal will not be given, since LISP never sees the cancelled line.) Any typing on a new line before the bell will be lost. If either $? or $WHAT? is typed back after input, the system is in Executive mode. Type " and repeat input.

2.2 SPECIFIC PROCEDURES FOR USING LISP

Initial LOGIN. If you are working at a location remote from SDC, your first step in using Q-32 LISP is to dial into the Time-Sharing System. This procedure varies, depending on the nature of the teletype hookup with the Q-32, and will not be described here. (See Part 1, Vol. 3 of the User's Guide.) Both remote users (when contact has been made) and local users then type in

```
        LOGIN xxxxx yyyyy
```

where xxxxx is the programmer's number, and yyyyy is the job number.

After LOGIN, the system will type back

```
$OK LOG ON n
```

where n is the teletype channel number.

LOAD Command. The next step is to request LISP to be loaded. The usual request is

```
        LOAD LISP
```

If there is sufficient space on the drum (approximately 47K is required), the system will type back

```
        $LOAD n
```

where n is the channel number, and you can proceed to the next step. If it types back

```
        $NO LOAD DRUMS FULL
```

then there is insufficient space on the drums, and you will have to repeat the procedure later. You can find how much space is available on the drum without loading, by typing at any time:

```
        ! DRUMS
```

If you wish to load a nonstandard private version of LISP, such as one on which you have some of your own function definitions, then the proper load command is

```
        LOAD ℓℓℓℓℓ nnnn
```

where ℓℓℓℓℓ is any unique name (6 alphanumeric characters) you choose, and where nnnn is the number of the tape on which your version is stored. The full description of the LOAD command is given in Part 1, Volume 4, of the User's Guide.

Occasionally, LISP (or your private LISP version) will not be stored on the Q-32 disc, but will be available only on tape. If the system typed back

```
        $WAIT
```

after the load request, then the system is not on disc, and you will have to wait for the operator to mount the correct tape. After the tape is mounted, you will receive either the

```
        $LOAD OK
```

or the

```
$NO LOAD DRUMS FULL
```

message. If you do not get a response to the load request within a few minutes, repeat it.

Combined LOGIN and LOAD Command. A new feature of the Executive allows the LOGIN and LOAD commands to be combined. You may type

```
LOGIN xxxxx yyyyy ℓℓℓℓℓ nnnn
```

where the four fields are as previously described. If you wish only to load LISP, then

```
LOGIN xxxxx yyyyy LISP
```

is sufficient.

The response of the Executive will be (like that to the LOAD command above)

```
$LOAD n.
```

GO Command. After LISP has loaded successfully, you should type in the command

```
GO
```

This command will cause LISP to start running and will also set the input mode to the object program mode (so that you should use the quotation mark when you want to cancel a line). LISP will type back the date and time, Q-32 LISP model number, and READY, and will then ring the bell. Don't type in any input until the bell rings.

LISP Input. After LISP has acknowledged the GO command with the typeout and the bell, you can type in pairs of S-expressions for Evalquote to execute. As in 7090 LISP, each pair is executed as soon as it is read, and the result is printed out. After the result is printed, the bell signals that LISP is again ready for input.

The conventions regarding the pairs for Evalquote are the same as those for 7090 LISP, with minor exceptions. The first element of the pair may be either a λ-expression or a function name. The set of available functions for Q-32 LISP is not the same as those available for 7090 LISP, but the most commonly used functions are available. Since Q-32 LISP is compiler-based, the effect of DEFINE is somewhat different from that in 7090 LISP. Defining a function via DEFINE causes the function to be compiled and the S-expression representation to be thrown away.

LISP Typing Conventions. The Q-32 LISP READ program is independent of line boundaries so that the last character of the nth line appears adjacent to the first character of the (n+1)th line. Consequently, a carriage return does *not* terminate an atomic symbol, and, if you want to type a message that ends with an atomic symbol, you must follow that symbol by a space. For instance, if Evalquote is given a function of no arguments, the second element of the Evalquote pair may be written either NIL or (). If NIL is used, it must be followed by a space; if the carriage return is typed without a space, the READ program will still look for a character to terminate the atomic symbol. (However, typing a blank on the next line will work.)

After Evalquote executes a pair, it throws away the remainder of the last line that it reads. Thus, if two complete Evalquote pairs are typed on the same line, the second one will be ignored. For the same reason, an excess of right parentheses at the end of the *second* element of an Evalquote pair will be ignored.

If you are typing in a long expression to Evalquote and you wish to cancel the entire expression, type the illegal percent character. The effect will be to cause an error in the READ function so that Evalquote will reject the entire pair. This trick is useful when you discover an error in a previously typed line and you want to begin the Evalquote pair over again.

The QUIT Command. After you have finished using LISP, you should terminate operations by typing in

```
! QUIT
```

This command will go to the Executive and will terminate your operation. It will also disconnect your teletype from the Time-Sharing System.

Reloading. Quite frequently, you may wish to reload without having to QUIT, LOGIN, and LOAD all over. This can be done by executing the LOAD command whenever necessary. You needn't QUIT first or LOGIN again as these are done automatically for you by the system.

SAVE Command. The time-sharing Executive now permits you to dump the current state of your core onto *disc* via the SAVE command. The form is

```
! SAVE ℓℓℓℓℓ
```

where ℓℓℓℓℓ if present is the 6 alphanumeric character name for this file and must be different from all the names currently on the disc. If this name is absent (blank), then the file will replace the one that you are currently using on disc (i.e., the one used in the LOAD command). The Executive will respond

```
$WAIT
```

while the dump is in progress, and then follow with

```
        $SAVE OK
```

when completed. If it replies $SAVE NG (NO GOOD) try another name.

Q-32 LISP has a similar SAVE feature, (see section 4.6) with the dump on tape rather than disc.

The RESCUE Feature. Certain LISP errors, such as illegal address references, can cause LISP to be interrupted by the time-sharing Executive. The RESCUE feature provides a mechanism whereby LISP can regain control and function properly. When such an interrupt takes place, the Executive gives control back to LISP in a specified location, and LISP will then call the unwind and backtrace procedures. If either unwind or backtrace itself causes an interrupt, the recovery procedure is halted, and LISP goes directly to the point where it looks for a new Evalquote pair. When a system error occurs, LISP will print

```
        RESCUE n
```

where n is a code indicating the nature of the interrupt. This printout may or may not be followed by a backtrace. Occasionally, a RESCUE printout may result from an illegal input to a LISP system function for which there is no diagnostic test; generally, the backtrace will make it clear that this is the difficulty. After the RESCUE printout is complete and the bell has rung, you can continue with your next Evalquote pair as usual.

Restart Procedure. On occasion, LISP may get itself into a state where the only way to get back to Evalquote is to return to the Executive program. In this case, you can try:

(a) Type

```
!STOP
```

to stop LISP and get back to the executive mode.

(b) Type

```
!$LIV/CR*
```

The system will then type back

```
!$LIV=n
```

where n is the location at which LISP stopped.

* Carriage return

(c) Type

```
40002'*
```

The system will acknowledge with

```
$MSG IN
```

(d) Type

```
/CR
```

The system should type back

```
n - 40002'    ; where n is the machine address for the symbolic
address $LIV.
```

If anything else comes back, repeat steps (b) through (d).

(e) Type

```
!GO
```

LISP will now act as though it had just started up. If it does does start properly, try typing a space followed by a carriage return. This step is occasionally necessary when LISP is hung up expecting input.

If LISP gets hung up in an output loop, it may be stopped by entering ! Blank CR (if you can get it in). To get going again, use the restart procedure above. Alternatively, you may press the "BREAK" key which induces a RESCUE interrupt.

If a LISP error is so bad that the procedure above does not work, then the only way to recover is to LOAD again. The effect of repeating LOAD will be to load a fresh copy of LISP. Errors requiring a fresh start can occur if the "garbage collector" becomes injured or confused; it will generally be evident when this is the case.

3. THE Q-32 LISP SYSTEM

This section describes the structure and operation of the Q-32 LISP system. Section 3.1 discusses reading and printing. Section 3.3 describes the overall structure of the LISP system in terms of the core map and data structures. Section 3.4 describes the working of Evalquote. Section 3.5 describes LISP Macros and the macro expander. Finally, section 3.6 discusses in detail LAP, the structure of the Pushdown List, and closed subroutines.

The description of all other functions, macros and special forms is left to section 4.

Sections 3.1 and 3.2 are of interest to all users of LISP. Sections 3.5 and 3.6 are useful for anyone who tries to write Macros or LAP code. Sections 3.3 and 3.4 are of interest mainly to advanced LISP users who wish to understand the system completely, modify the system or to make functions which manufacture other functions.

3.1 READING AND PRINTING

Q-32 LISP read functions READ and READTAPE can be made to accept the following character set from teletype or magnetic tape card image:

Letters:

A through Z
* / $ =

Delimiters

Space , ( ) .

Numerics

+ - Ø through 9

Illegal Characters

: > # % \ ← Bell] ; ↑ ? [<

Ignored characters

Line Feed Carriage Return

Special treatment

' (prime)

Atomic symbol is a number or any string of letters and numerics starting with a letter and terminated by a delimiter, or else an atom input in one of the following manners:

1) The $$ artifact permits any arbitrary string of characters to be inserted into LISP as an atomic symbol.

   $$@ (string of characters not containing @)@ (where @ is any character) is used to insert any string of characters as an atomic symbol.

   Example: $$/%))/ inserts the atom %)) into LISP

2) Outside of a $$ artifact, the special character ' (prime) followed by any character except carriage return is converted to a character atom whose representation is the address 1Q4 plus the octal representation of the character. Thus, 'A (A corresponds to 21Q) becomes address 10021Q. If ' (prime) is followed immediately by a carriage return, the carriage return is ignored and the first character from the next line is used to form the character atom. ' (prime) is not a delimiter, and hence must be set off on the left by a blank, comma, left or right parenthesis, dot, or another character atom, or else all characters to the left of the ' (prime) will be lost.

Thus, '' means character atom corresponding to '

'1 means character atom corresponding to 1, etc.

Any character at all may appear in the $$ string except the Carriage Return and Line Feed characters, which are ignored.

A number is one of the following forms:

integer	1	12	+2E4	-35
octal integer	27Q	27Q3	-14Q	-14Q5
floating point number	1.0	-Ø.5	+1.75	224.
	2.0	+357.75E-3		

where E is the power-of-ten scaling of scientific notation for integers and floating point numbers; and Q is the power-of-eight scaling for octal numbers. Octal and integer numbers may have only positive scalings.

Note: A number must start with a numeric. It can contain at most one decimal point. It may contain the letter E.

All numbers are converted on input to one of three internal representations: integer, octal, or floating. On output, the numbers are then reconverted.

Floating number input and output conversions give 11 to 12 significant figures. Floating point printout is 12 places of which the twelfth may be in error by ± 4.

If the read program encounters an illegal character outside of a $$@ string, then ERROR is called and the current program is unwound.

Tape Reading and Printing

Tape reading uses the first 72 columns and ignores the last 8 columns of the card image to allow for sequencing.

Printing using PRINT or PRINTAPE use a "pretty-print" formatting logic which:

1) prints one S-expression, supplying parentheses and dots as required,

2) prints numbers according to their internal coding:

```
  1000Q prints as 1Q3
    512 prints as 512
   1.76 prints as 1.76ØØØØØØØØ           Some of these zeros
   22.0 prints as 22.ØØØØØØØØØ           may be suppressed
 2234.5 prints as 2.2345ØØØØØØØE3
```

3) If the expression will not fit on a 72 character line, the line is broken as follows:

- after the first RPAR at the lowest parenthesis depth at which the line can be broken and still fit;
- after the last atom or symbol which will fit, if no RPAR is found;
- in the middle of an atom as a last resort if no other break point can be found, e.g., an atom consisting of 73 characters.

If the line is broken, the next line starts with n spaces, where n is the parenthesis depth, and the line-breaking algorithm is applied to the new line, except that if an atom has to be broken, or if the parenthesis depth exceeds 70, no indentation is used in printing.

3.2 TYPES OF VARIABLES

In Q-32 LISP, a literal atom can have one of two statuses, Special or Unspecial, governed by a flag in bit 2 of the atom head (see section 3.3).

Local Variables: If an atom which is not Special is bound in a function by PROG or LAMBDA, then the atom is regarded strictly as a local dummy variable. Within the lexical scope of the PROG or LAMBDA the atom name is simply an address on the pushdown list. If it is a LAMBDA variable it is bound initially by the function call, and may be reset within the function (viz. by SETQ). If it is a PROG variable, it is set to NIL at the entrance to the PROG, and may be reset by functions inside the PROG, but it is invisible outside of the scope of the PROG in which it is bound. The lexical scope of a function (i.e., the variables bound by LAMBDA) includes any PROG found within the LAMBDA but excludes any LAMBDA expression within functional arguments in the function. A local dummy variable is meaningful only at compile time.

Special Variables: An atom which is in Special status always retains its atomic identity. If used as a free variable, it has the lowest level binding applicable at the time of use. If a special atom is bound by LAMBDA, the following results occur:

1) At the time of entry into the function, the old value of the variable is saved on the PDL.

2) The new value of the variable is stored in the special cell of the atom replacing the old.

3) All changes to the value of the variable are made in the special cell.

4) At exit from the function, the old value of the special variable is recovered from the PDL and restored to the special cell of the atom.

If the special atom is bound by PROG, the same steps occur, except that the new value of the variable at entrance to the PROG is, as usual, NIL.

From the above it can be seen that a special variable, when used as a free variable, always shows its most recent binding. (The atoms of 7090 LISP are thus more similar to the Special atoms of Q-32 LISP than they are to Unspecial atoms of Q-32 LISP.)

Setting of Free Variables Zero-Level Bindings: An atom is considered to have a zero-level binding if it can be used completely free (i.e., not bound by LAMBDA or PROG), and acts like a constant. In Q-32 LISP, there is no APVAL mechanism, and the zero-level binding is done directly in the atom head. The only exceptions are the atoms T and F which are treated as special cases by the compiler and cannot be bound by LAMBDA or PROG. (T always evaluates to quote T and F always evaluates to quote NIL.) Numbers, character atoms, and NIL cannot be bound.

Zero level bindings of atoms and the current binding of Special atoms in Q-32 LISP are stored in the CAR of the atom head. For unbound atoms, the CAR of the atom head points to NIL (=∅). For atoms which have a functional binding, the CAR points to the first cell of the compiled code for the function. All other special bindings are made indirectly through a special cell pointed at by the CAR of the atom head (see section 3.3 for examples).

The functions SETQ, CSETQ and CSET may all be used to change zero-level bindings of free variables, but SETQ cannot be used to establish a zero-level binding and will cause serious errors* if applied to an atom which is either unbound or has a functional binding.

Really, CSETQ and SETQ are identical except:

1. CSETQ makes variables Special
2. SETQ returns value of 2nd Arg., CSETQ 1st Arg.

If a variable is used completely free, i.e., is bound only at zero level, the action of CSETQ and SETQ are identical, except that CSETQ changes the status of the variable to Special at run time and SETQ does not. Also, SETQ returns 2nd Arg, CSETQ 1st Arg., as value. For variables already in Special status, CSETQ and SETQ produce identical results.*

* For Special variable X, (CSETQ X Y) is equivalent to (RPLACA (QUOTE X) (LIST Y)), while (SETQ X Y) is equivalent to (RPLACA (CAR (QUOTE X)) Y). The second form produces undesirable effects if(CAR (QUOTE X)) is not a true list pointer.

Free Use of Unspecial Variables: Use of Unspecial variables as free variables causes the compiler to give a message of the form (variable NOT DECLARED) but does not prevent correct compilation, since the compiler handles the variable as if it were Special. However, if at run time the unbound variable has no Special cell and the function tries to set the variable, a serious error will be induced.

Functional Argument: Atoms bound by LAMBDA may be used as functional variables in Q-32 LISP with no difficulty. It is not necessary to declare the functional variables Special, since the compiler recognizes them by context.

In calling a function which requires a functional argument, FUNCTION must always be used. QUOTE will not work, since the calling function requires the special binding of a functional argument, not the name of a function, as described below.

FUNCTION can be used with a simple (atomic) function name or with a LAMBDA or LABEL expression. In the atom case, FUNCTION causes the special binding of the atom (the CAR of the atom head) to be passed to the calling function. Thus (FUNCTION FN) acts like FN, not like (QUOTE FN). (A simple function name can be used without FUNCTION. This will cause the compiler to print out (FUNCTION NOT DECLARED) but it will work.) In the case of a LAMBDA or LABEL expression, FUNCTION causes the functional argument to be compiled at compile time into a subdidiary function and passes the pointer for the subsidiary function to the calling function, so that the LAMBDA or LABEL expression case is reduced to the simple function case.

Examples:

1) Special variables required: X and Y must be declared Special in order to make the definition of SUBST1 work. Note also use of FUNCTION:

```
SPECIAL ((X Y))
DEFINE (((SUBST1 (LAMBDA (X Y Z)
     (MAPCAR Z (FUNCTION (LAMBDA (J)
       (COND ((EQUAL Y J) X) (T J))) ))) ))))
UNSPECIAL ((X Y))
```

2) Zero level binding of free variable.

```
CSET (PI 3.14159)
(LAMBDA (X) (TIMES X PI)) (2)
result 6.28318   PI is Special
```

3) Restoring of higher level bindings, assume PI set as above

```
(LAMBDA (PI) (CSET PI 5)) (B)
```

result = 8 B is set to 8 and PI is still 3.14159. Once B has been bound, SETQ works like CSETQ.

```
(LAMBDA (PI) (CSET PI B)) (PI)
```

result = PI. This somewhat confusing example is intended to show that an atom cannot be set at zero level if it is bound by LAMBDA or PROG. In operation of this function, the atom PI is first bound to quote PI, then to 8, the value of B, but at the end is restored to its original value of 3.14159.

```
(LAMBDA (X) (SETQ PI X)) (ø)
```

result = ø This time PI is changed to a new value ø.

4) Another example of FUNCTION

```
(LAMBDA (X) (MAPCAR X (FUNCTION SUB1))) ((ø 1 2 3))
```

result = (-1 ø 1 2)

3.3 DATA STRUCTURE IN Q-32 LISP

Storage Allocation: The Q-32 LISP system occupies octal locations 4øøøøQ to 172777Q in core and has a total length of 46592 (decimal) cells. As shown in Fig. 1, the space is divided into six areas:

Binary Program Space. Binary Program Space starts at 4øøøøQ and may run up to 74776Q. The reserved atom TBPS (mnemonic for Top of Binary Program Space) points to cell 74777Q. The reserved atom BPORG points to the next available cell in Binary Program Space, and the reserved atom *BPORG is used to back up in case of error. DEFINE, LAP and MACRO compile code for all functions or macros into Binary Program Space.

Scratch Program Space. Scratch Program Space starts at 75øøøQ and may run up to 75777Q. The cell 76øøøQ, which is the origin of the Pushdown List, is protected against being overwritten from Scratch.

Scratch Program Space is used by Evalquote and the functions EVALQT and *EVALQT to compile code for interpreting all functions, Macros and special forms which cannot be operated directly.

Octal Address		Remarks
4ØØØ	Binary Program Space	BPORG is at 6ØØ52 in LISP 1.5 M2.5
74777	Protected Cell	TBPS points here
75ØØØ	Scratch Program Space	*SCRACH points here *SCRACHX points to next available cell
75777		
76ØØØ	Pushdown List	Pushdown list origin-protected from Scratch programs
		Index Register 1 points to start of next available block of PDL.
1Ø17ØØ		Protected from Index Register 1
1Ø1777		
1Ø2ØØØ		OBLIST atom head
	Atom Heads and Quote Cells	
1Ø7777		
11ØØØØ	Full Word Space	
172575	Free Storage	
172576	172577	OBLIST Special Cell
172577	172600	Pointer to OBLIST array
1726ØØ	OBLIST buckets	
172775		

Fig. 1 Storage Allocation in LISP System

Pushdown List. The Pushdown List (PDL) starts at location 76ØØØQ and may run up to location 1Ø1777Q. The pushdown block for a function uses as many cells as the function requires. Index register 1 always points to the start of a pushdown block and is saved and changed by a function to protect its block before it calls another function. Since the called function could in principle use up to 32 arguments or 1ØØQ cells, Index Register 1 is protected from exceeding 1Ø17ØØQ.

The Pushdown List is used to store all arguments, program variables, temporary pointers, return addresses, and also the previous values of special variables which are used as program variables or function variables. The structure of the PDL is described in section 3.6.

Atom Heads and Quote Cells. The region of core from 1Ø2ØØØ Q up to 1Ø7777Q is reserved for quote cells and for atom heads for (unique) literal atoms. OBLIST is at 1Ø2ØØØQ. A quote cell is a single cell containing only a CAR pointer to an atom or to a piece of list structure. It is assigned uniquely by LAP (e.g., two different references to (QUOTE (A B C)) point to the same quote cell), is never collected by the garbage collector, and serves to protect the list structure, atom or number to which it points. All references from binary program space to LISP data, i.e., numerical, literal or list structure constants, are made via quote cells.

An atom head for a literal atom contains an atom head flag (bit number 1 = 1) and a CDR pointer to a pointer to its print name and property list. The CAR is either NIL or a special binding. Bit number 2 of the atom head indicates an atom in Special status (see section 3.2 for meaning of Special status, and a later paragraph in this section for further description of Word Use). Atom heads in Q-32 LISP are not protected by the OBLIST, but may be collected by the garbage collector under certain circumstances. If the free cells in the atom head or quote cell area are exhausted, the garbage collector reclaims all atom heads which are not in Special status, are not pointed to, have no binding and have no property list. (The print name is not considered a property in Q-32 LISP.)

Full Word Space and Free Storage Space. Full Word Space and Free Storage occupy the region from 11ØØØØQ to 172775Q jointly. Full Word Space starts at 11ØØØØQ and is filled downward in Figure 1 (i.e., toward higher numbered registers). It is used to store arrays. These include print name arrays for literal atoms and arrays for storing LISP numbers. Free Storage starts at 172577 and is built upwards, except that the OBLIST buckets occupy the area from 1726ØØQ to 172775Q. CONS adds one cell to the top of Free Storage. When Free Storage and Full Word Space meet, the garbage collector is called to compact Full Word Space upwards and Free Storage downwards. Free Storage is used only for storage of list structures.

Word Use. The Q-32 data word consists of 48 bits, divided up into four parts, the prefix, decrement, tag and address, which occupy bits Ø-5, 6-23, 24-29, 3Ø-47 respectively. Core locations in binary program space and in scratch space hold instructions and data cells corresponding to compiled programs. Cells on the Pushdown list in general hold address pointers as described in section 3.6.

The use of words within the Atom Head and Quote Cell Area, Free Storage and Full Word Space is shown in Fig. 2. Within Atom Head and Quote Cell Space and Free Storage spaces bit Ø of the prefix is used by the garbage collector, bit 1 is always an atom flag (the function ATOM tests this bit only), bit 4 is used as a flag indicating an atom in Special status, and the remainder of the prefix is unused. The tag portion of the word is unused except for number pointers (which are like atom heads for numbers but are in Free Storage, are non-unique, and are never Special). For number pointers, the tag is 71 for integers, 72 for floating point numbers and 75 for octal numbers.

A Quote Cell has only a single address pointer in its CAR (the CDR is always NIL).

Full Word Space is used for storage of arrays. Each array has an array head cell followed by a contiguous block of core containing the array.

Within the array head, bit Ø is used by the garbage collector, bits 1, 2 and 3 are unused. Bit 4 is 1 if the array contains non-list type data (at present the only allowed type). Bit 5 is zero for numerical arrays, 1 for BCD data. The tag is used for BCD arrays to indicate the number of characters in the last word (left justified).

The decrement is used to indicate the number of words in the array, exclusive of the array head. For LISP numbers this is always 1, while for Pname arrays the length is essentially unlimited. The CAR of an array head always contains a back pointer to the cell in Free Storage which points to the array.

LISP Word	Word Part	Prefix	Decrement			Address	Core Locations
	Bit	0 1 2 3 4 5	6 through 23	24-26	27-29	30	
Atom Head		G 1 X X S X A	Pointer to Pname pointer	Ø	Ø	Ø if unbound, or pointer to Special Binding	1Ø2ØØØQ through 1Ø7777Q
Quote Cell		G Ø Ø Ø Ø Ø A	Ø	Ø	Ø	Pointer to Atom Head or List Structure	
Pname or Literal Array Pointer		G Ø Ø Ø Ø Ø A	CDR-Pointer to Atom Head, Quote Cell, Free Storage, or NIL (=Ø)	Ø	Ø	Pointer to Literal Array	Free Storage above Full Word Space and below 172776Q
Number pointer		G 1 Ø Ø Ø Ø A	Ø	7	N	Pointer to Number Array	
List Structure pointer		G Ø Ø Ø Ø Ø A	CDR-pointer to Atom Head, Quote Cell, Free Storage, or NIL (=Ø)	Ø	Ø	CAR-pointer to Atom Head, Quote Cell, Free Storage or NIL (=Ø)	
Numerical Array Head		G Ø Ø Ø 1 Ø L	(one word array) 1	Ø	Ø	Pointer back to Number Pointer	Full Word Space 11ØØØØQ to above Free Word Space
Numeric Array cell		Numerical value					
Pname or Literal Array		G Ø Ø Ø 1 1 L	Number of array words	Ø	B	Pointer back to literal array pointer	
Pname array cell		Pname Hollerith left justified filled with octal 77's					

LEGEND:

G - Bit normally zero, used by garbage collector

S - Ø if not special status; 1 if special status

L - Ø for pointer arrays (not implemented);1 for non-pointer arrays

A - 1 indicates atom; Ø if not atom

N - 1 for integer; 2 for floating point; 5 for octal

B - Number of characters (1-8) in last word of Pname array

X - Unused (Ø)

Fig. 2 Use of Word Within Q-32 LISP

Atom and List Structures. Fig. 3 shows five examples of atom and list structures in Q-32 LISP.

In Example 1, the atom DIFFERENCE, the name of a simple function, is shown. Its atom head in the Atom Head and Quote Cell area has an atom flag but not a Special flag. Hence the prefix is octal 20. The CAR of the atom head points into BPS to the start of code for DIFFERENCE. The CDR prints through a pointer cell in Free Storage to a two-word-plus-array-head Pname array in Full Word Space. The array head has an Ø3 Prefix to indicate a non-list type literal array. The 2 in the CDR of the array head shows a 2 word array, and the Ø2 tag shows that the last word contains 2 left-justified BCD characters. The CAR of the array head points back to the pointer cell in Free Storage. During garbage collection, the Pname array and the Free Storage pointer all may be moved, but the Atom Head will remain fixed.

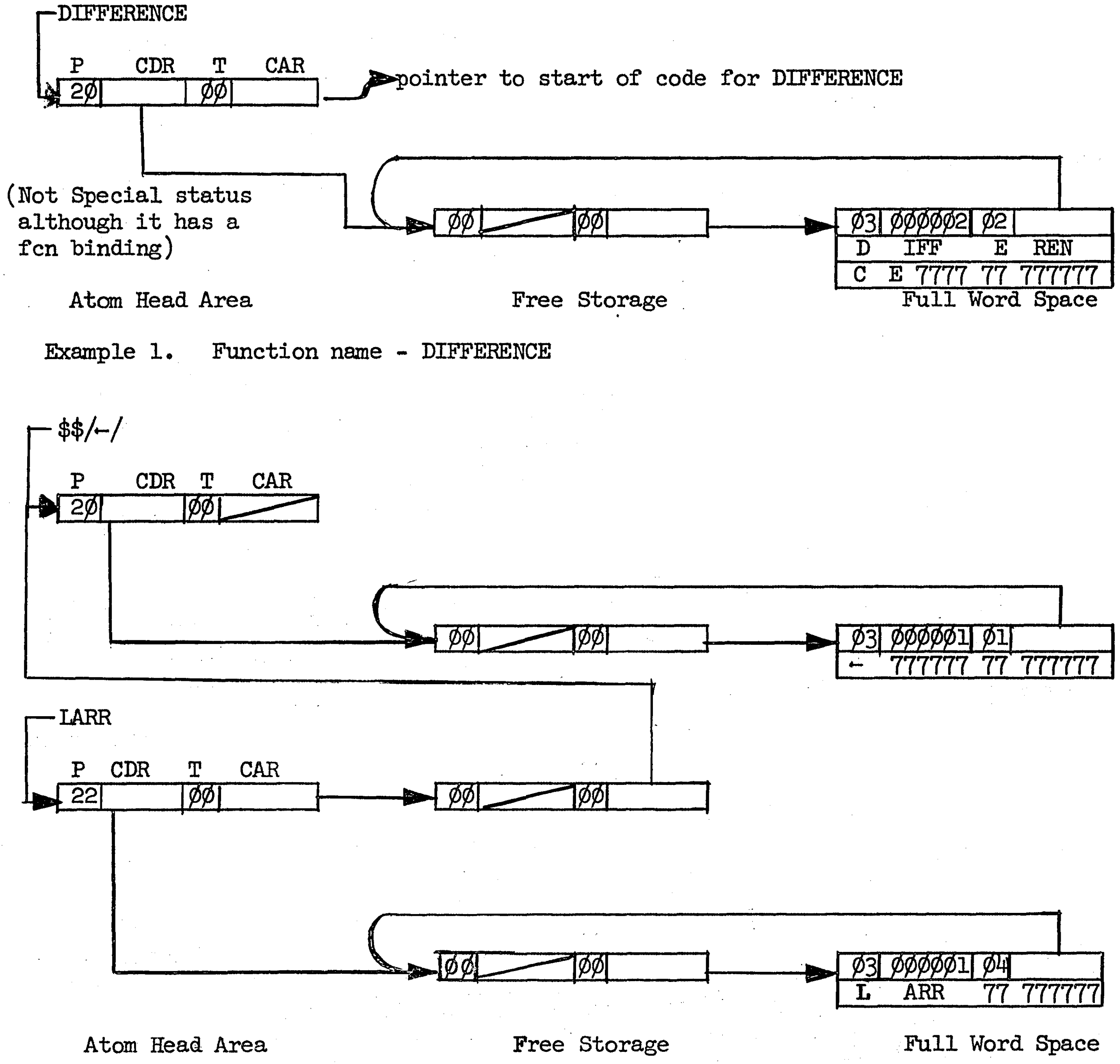
Example 2 shows the result of setting the atom LARR to the zero level binding $$/←/ . The atom ← is generated Unspecial and unbound, and is pointed to indirectly from the CAR of the atom head for LARR. LARR is in Special status.

Example 3 shows a LISP number, the integer 1Ø1 (decimal). Its atom head, which is non-unique, is a cell in Free Storage which has the prefix 2Ø to indicate an atom, and a tag of 71 indicating a decimal integer. The CAR of the Free Storage cell points to an array of one cell whose array head has a Ø2 Prefix to indicate a numerical array, a CDR of 1 to indicate one data cell, and a CAR pointer back to Free Storage.

Example 4 shows an atom with a property list, in this case the Macro PLUS. The CAR of the atom head points to location Ø designating no zero level binding. The CDR points to a list of three elements whose CARs point to the Pname for PLUS, to MACRO, and to the start of the code for expanding PLUS expressions.

Example 5 shows a quoted list of three elements. Its Quote Cell in Atom Head and Quote Cell area is zero except for its CAR which points to the list in Free Storage Area. The list in Free Storage is made of pointers to atoms A, B and C respectively. The quote cell can never be collected by the garbage collector. The list in Free Storage is protected from collection by being pointed at by the quote cell, and atoms A, B and C are protected by the quoted list in Free Storage.

Garbage Collector. The garbage collector in Q-32 LISP is designed to compact and collect lists and arrays, and to collect those Gensyms and other atoms that are not being used (i.e., not bound at any level, not pointed to, and having no property list). The object list is used mainly as a dictionary for atoms, and if atom cell space is short, the OBLIST is not used to protect atoms. Quote cells will protect atoms, lists and arrays, and LAP sees that quote cells are never duplicated, by searching through all existing quote cells, using the EQUALN test, before establishing a new quote cell.

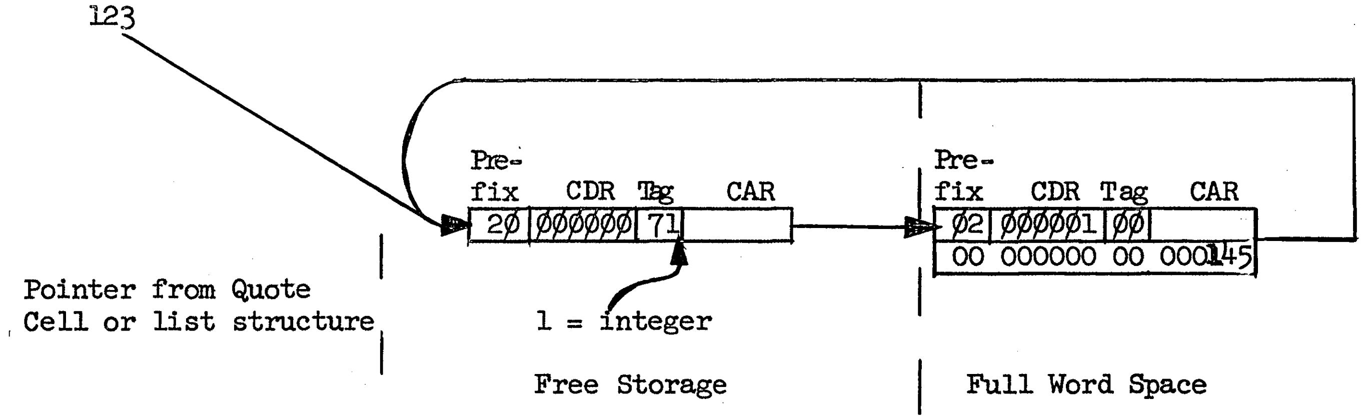


Example 1. Function name - DIFFERENCE

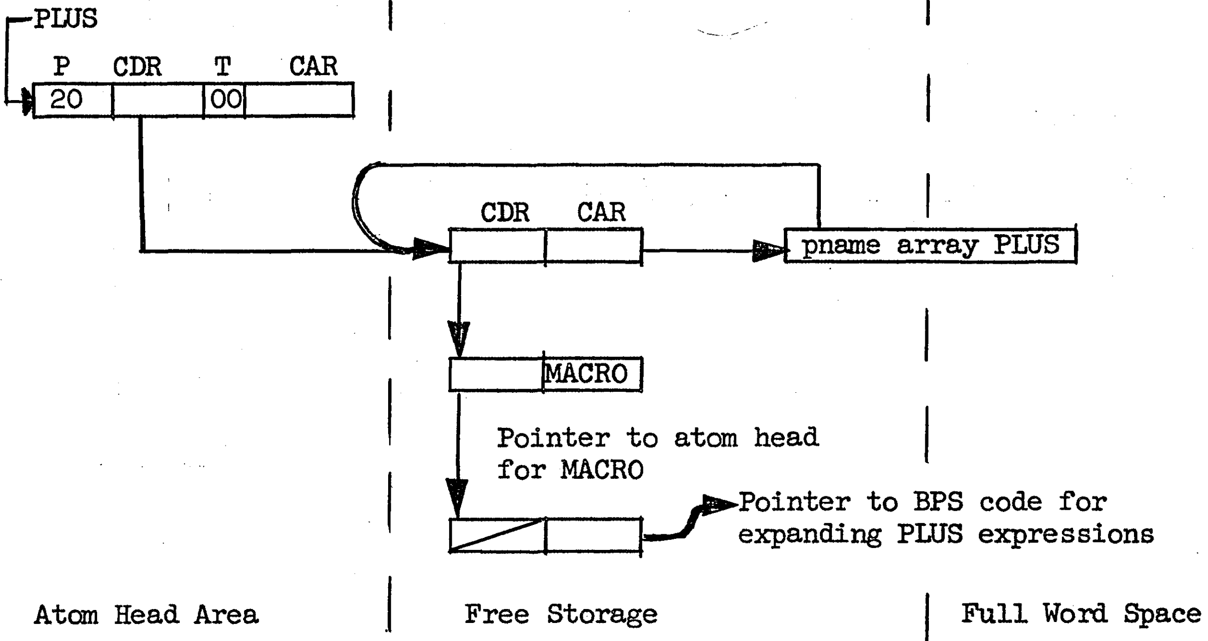
Example 2. CSET (LARR $$.←.) LARR is bound and Special
← is unbound and not Special

Fig. 3 Examples of Q-32 LISP Structures

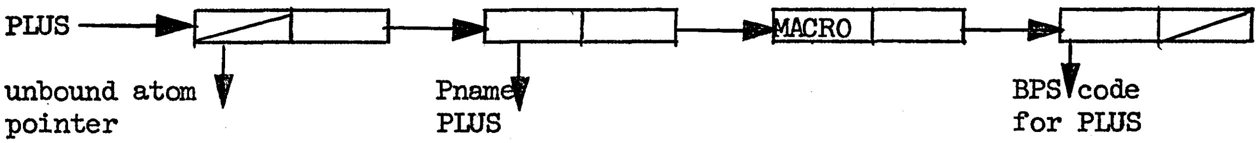
Fig. 3 Examples of Q-32 LISP Structure (Contd.)



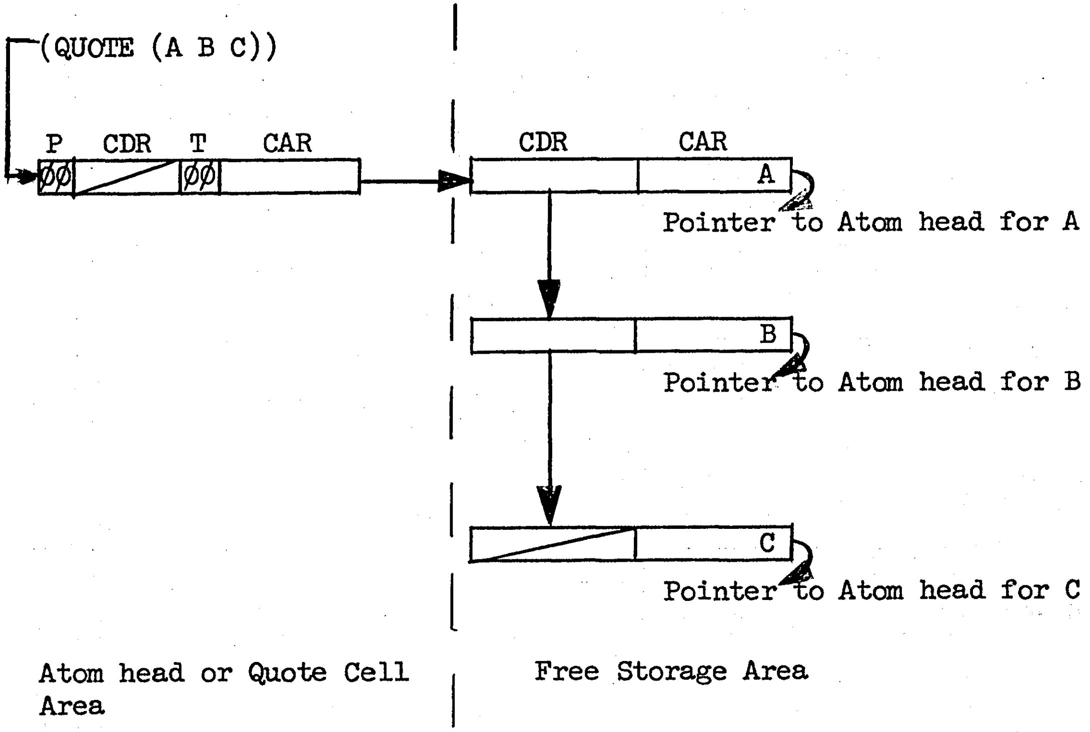
Example 3. The LISP Number 101 (integer)



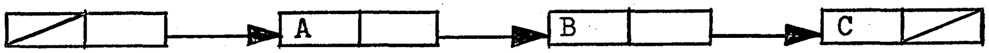
in Standard LISP notion, this is equivalent to



Example 4. An atom with a property list-the Macro PLUS


(QUOTE (A B C))
P
CDR
T
CAR
ØØ
ØØ
CDR
CAR
A
Pointer to Atom head for A
B
Pointer to Atom head for B
C
Pointer to Atom head for C
Atom head or Quote Cell Area
Free Storage Area

in standard LISP notation, this is shown as



Example 5. A quoted list

Fig. 3 Examples of Q-32 LISP Structures (Contd.)

Atoms are always preserved if they are in Special status, have properties, or if they possess Special bindings. They are also preserved if they are pointed to from a protected area. Pointers in Binary Program space go only to quote cells or to atom heads of atoms with Special bindings. Hence, Binary Program space is not used for marking.

The OBLIST. The OBLIST is a pointer to a series of 125 buckets from each of which hangsa list of pointers to literal atoms. The bucket from which an atom hangs is determined by a simple hash coding scheme based upon the first word of the print name. The OBLIST is used by the read programs as a rapid look-up table for atoms. Whenever atoms are collected by the garbage collector, the OBLIST is discarded and the remaining literal atom print names are bucket sorted again and restrung to create a new OBLIST.

Garbage Collection. Garbage collection is done in a five phase process:

1. All list structure is marked, starting from the quote cells, the object list or selected atom heads and the pushdown list. Full words are marked with a bit in the array head, so a bit table is not required.

2. Full word space is compacted downward. Two pointers start at the beginning of full word space. The first pointer is advanced over all full words, and those marked are copied into the location indicated by the second pointer, which is advanced for each array copied. The pointer in the array head is used to update the list pointers to relocated arrays.

3. Free storage is compacted upward by a scheme attributed to D. Edwards. Two pointers are set, one to the top of free storage and one to the bottom. The top pointer scans words, advancing downward, looking for one not marked. When one is found, the bottom pointer scans words, advancing upward, looking for a marked word. When one is found, it is moved into the location identified by the other pointer. A pointer is left at the location the word was moved from, pointing to where it was moved to. The top pointer is then again advanced as before. The process terminates when the two pointers meet.

4. List references to the vacated free storage are fixed up by looking at CAR and CDR of every word on the pushdown list, on the OBLIST, and in the compacted free storage. Any pointers to the vacated area are replaced with pointers to the relocated words, using the pointers left there in step 3.

5. The OBLIST is re-created if it was not used for marking atoms by performing a bucket sort on the print names of the remaining literal atoms.

3.4 EVALQUOTE

Evalquote used in the Q-32 LISP works as follows: It takes two arguments, the first being a function name, the name of a Macro or special form, and the second being a list of arguments to be regarded as quoted arguments for the function. It is possible, of course, to have a special form beginning with LAMBDA or LABEL or PROG or to have (with proper caution being observed) an expression which, when evaluated, will produce a function descriptor. If Evalquote finds that the first argument is an atom and is a bona fide function, it passes to the ultimate evaluator function *EVQ two arguments, the first of which is a pointer to the machine code for the function. A second argument is the list of arguments originally given to Evalquote.

In all other cases, namely where the first argument for Evalquote is a Macro or special form or something which is not an atom, a LAMBDA expression is concocted and fed to the compiler under the name *FUNC. *FUNC is compiled into a scratch area which is reused every time Evalquote has to compile. Finally, Evalquote calls *EVQ with the arguments CAAR of *FUNC and either the original argument list that was given to Evalquote or, if necessary, an argument list to which has been appended a list of all quoted variables which occurred within the first argument that was given to Evalquote. The function descriptor is modified to accept these arguments as values for additional variables. These are used instead of quote cells to prevent irreparable loss of quote cell space. (Once a quote cell is created it can never be collected.) Once *FUNC has been produced, the original function descriptor can be used again by giving Evalquote a first argument *FUNC, until *FUNC is recompiled by the next non-atom or non-function encountered by Evalquote. Evalquote is not a function and so is not callable within the system. However, the system includes the callable function EVALQT, which is a function of the same two arguments that Evalquote takes at the top level. When EVALQT is called, the same thing happens as when Evalquote itself is called at the top level except that if compilation has to occur, GENSYM's are used to name all compiled functions. Upon return from EVALQT, these GENSYM's are unbound so that they can be garbage-collected.

3.5 MACRO AND THE MACRO EXPANDER MDEF

Q-32 LISP contains a provision for defining Macros using the function MACRO and a Macro expander, MDEF, for expanding Macros before a function is compiled. The general flow through the system, and a typical example, is shown in the following example:

Stage	Status	Remarks
S-expression input to DEFINE	(PLUS3 (LAMBDA (A B C) (PLUS ~~(~~A B C)))	PLUS is a Macro
↓ Processed by Macro expander MDEF		
S-expression input to Compiler	(PLUS3 (LAMBDA (A B C) (*PLUS A (*PLUS B C))))	*PLUS is a function of 2 arguments
↓ Compiler and LAP		
	Binary Code for PLUS3	

A Macro is a function of one argument which is applied to an S-expression before compilation and without evaluation of the S-expression. The argument of the Macro is the entire form containing the Macro, i.e., the S-expression whose CAR is the name of the Macro. (In the above example, the Macro PLUS is applied to the argument (PLUS A B C) .)

In order to define a Macro in Q-32 LISP, one writes the expression for a function of one argument and gives it to the function MACRO rather than DEFINE.

MACRO causes the definition to be compiled into binary code, just as DEFINE would for a function. After the compilation is completed, MACRO then attaches the pointer to the binary code on the property list of the name of the Macro under the property MACRO. (For a function, the pointer would be placed in the CAR of the function name.) The property MACRO is used by the Macro expander MDEF to obtain the code for expanding a Macro.

Examples:

1. A simple Macro (not in standard Q-32 LISP) is FLAMBDA, defined as follows:

```
MACRO (((FLAMBDA (LAMBDA (L)
          (LIST (QUOTE FUNCTION)
            (CONS (QUOTE LAMBDA) (CDR L)))))))
```

which converts any S-expression of the form

```
(FLAMBDA (Args) expression)
```

into the form

```
(FUNCTION (LAMBDA (args) expression))
```

2. The Macro definition of IF is the following:

```
MACRO (((IF (LAMBDA (L)
       (CONS (QUOTE COND) (LIST (CADR L) (CADDR L))
         (COND (QUOTE T) (CADDDR L)))))))
```

which converts an expression of the form

```
(IF p q r)
```

into the form

```
(COND (p q) (T r))
```

3. The Macro PLUS is defined as follows:

```
MACRO (((PLUS (LAMBDA (L) (*EXPAND L(QUOTE *PLUS))))))
```

where *PLUS is a function which adds its two arguments, and the function *EXPAND is defined in the system by the expression:

```
DEFINE (( (*EXPAND (LAMBDA (L OP) (COND ((NULL (CDDR L)) (CADR L))
(T (LIST OP (CADR L) (CONS (CAR L) (CDDR L)))))))))
```

If the expression (PLUS 2 3 4 5) is encountered, MDEF will change it to

```
(*PLUS 2 (PLUS 3 4 5))
```

and MDEF will then be applied recursively to this expression until the expression is expanded to the form

```
(*PLUS 2 (*PLUS 3 (*PLUS 4 5)))
```

4. Macros can also be used to form functions that quote their arguments as well as functions of an indefinite number of arguments.

For example, a Macro that would quote all of the list containing it could be defined by

```
MACRO (((QUOTEF (LAMBDA (L) (COND ((NULL (CDDR L))
 (CONS (QUOTE QUOTE) (CDR L)))
 (T (LIST (QUOTE QUOTE) (CDR L))))))))
```

Then (QUOTEF A B C D) would yield (QUOTE (A B C D)) while (QUOTEF A) would yield (QUOTE A)

One will note that any function name that is to appear in the output macro expansion must be quoted inside of the Macro definition. Macros and functions can be freely mixed inside LISP expressions. However, since Macro expansion occurs at compile time, a Macro must always be defined before its name is used, and changing a Macro definition has no effect on previously defined functions or on the other Macros. (For functions, on the other hand, the definition existing at call time is the one that counts.)

3.6 LAP, PUSHDOWN LIST, CLOSED SUBROUTINES

LAP:
LAP is a two-pass assembler. It is used by the LISP compiler, but it can also be used for defining functions in machine language, and for making patches. LAP is an entirely internal assembler. Its input is in the form of an S-expression that remains in core memory during the entire assembly. No input or output occurs. It assembles directly into memory during the second pass. LAP can be used just like any other LISP function; however, since the effect of LAP is to compile code or place a binary patch into core, and the value of LAP is not usually of interest to the user, LAP is usually operated at the highest level for the effect it produces.

Format:
LAP is a function with two arguments. The first argument is the listing, the second argument is the initial symbol table in the form of a list of dotted pairs of the form (symbol . value). The value of LAP is the final symbol table.

The first item of the listing is always the origin. All remaining items of the listing are either location symbols if they are atomic symbols other than NIL, or instructions if they are composite S-expressions.

Origin:
The origin informs the assembler where the assembly is to start and whether it is to be made available as a LISP function. The origin must have one of the following formats:

- If the origin is an octal or decimal number, then the assembly starts at that location.

- If the origin is an atomic symbol other than NIL, then this symbol must have a zero-level binding to a number that specifies the starting location.

- If the origin is NIL, then the assembly will start in the first available location in the binary program space. If the assembly is successfully completed, then the cell specifying the first unused location in binary program space is updated. If error diagnostics are given during compilation, the binary program counter (BPORG) will not be moved.

- If the origin is of the form (NAME SUBR n), where n is the number of arguments that the subroutine expects, then assembly is in binary program space, as in the case above. If the assembly is successful, the CAR of the name will be made to point to the origin of the program. If the assembly is not successful (if any error diagnostic has occurred), then the atom name will point to wherever it was pointing before and BPORG will be left pointing to the start of the program that was compiled, so that the next compilation will clobber it.

Symbols:
Atomic symbols appearing on the listing (except NIL or the first item on the listing) are treated as location symbols. The appearance of the symbol defines it as the location of the next instruction in the listing. During pass one, these symbols and their values are made into a pair list, and are appended to the initial symbol table to form the final symbol table. The final table is used in pass two to evaluate the symbols when they occur in instructions. It is also the value of LAP.

Symbols occurring on this table are defined only for the current assembly. The symbol table is discarded after each assembly.

Instructions:
Each instruction is a list of from one to four fields. Each field is evaluated in the same manner; however, the fields are combined as follows:

- The first field is taken as a full word.

- The second field is reduced algebraically modulo 2^{18} and is OR'ed into the address part of the word. An arithmetic -ØQ is reduced to 777777Q.

- The third field is shifted left 18 bits and then OR'ed into the word.

- The fourth field is reduced modulo 2^{18} and is shifted left 24 bits and OR'ed into the decrement.

Fields:
Fields are evaluated by testing for each of the following conditions in the order listed:

- If the field is atomic:

    a. The atomic symbol NIL has for its value the current origin of binary program space. During an assembly that is not in binary program space, this cell contains the starting address of the next assembly to go into binary program space.

    b. The atomic symbol $ has the current location as its value.

    c. The symbol table is searched for an atomic symbol that is identical to the field.

    d. If the field is a number, then its numerical value is used.

- If the field is of the form (E a), then the value of the field is the address of the S-expression a, which should be a literal atom.

- If the field is of the form (QUOTE a), then a quote cell pointing to a in the address is created (if it does not already exist). It is the address of the quote cell that is assembled. Quoted S-expressions are protected against being collected by the garbage collector. A new literal will not be created if it is EQUALN to one that already exists.

- If the second field is of the form (*SPECIAL x), then the value is the CAR of the atom x. The assembled instruction contains the address of atom x, with the indirect bit set in the tag field (equivalent to a tag of 20Q).

- In all other cases, the field is assumed to be a list of subfields, and their sum is taken. The subfields must be of types listed above.

The set of operations codes that are recognized by LAP is :

Octal code	Mnemonic	Name
014	BUC	Branch Unconditionally
020	SFT	Shift
050	FST	Full Store
051	STZ	Store Zero
100	ADD	Add
110	SUB	Subtract
120	MUL	Multiply
134	DVD	Divide
200	LDA	Load Accumulator
204	LDM	Load Magnitude (Accumulator)
210	LDC	Load Complement(Accumulator)
220	LDB	Load B Register
230	LDL	Load Logical Register
300	FAD	Floating Add
310	FSB	Floating Subtract
320	FLT	Float
324	FRN	Floating Round
330	FMP	Floating Multiply
334	FDV	Floating Divide
400	CAS	Compare Accumulator with Storage
420	LDX	Load Index (Register)
43003	XOR	Exclusive OR
424	ATX	Add to Index (Register)
430	CON	Connect Accumulator with Storage
434	LDS	Load and Shift
500	STA	Store Accumulator
504	STB	Store B Register
510	STL	Store Logical Register
520	STX	Store Index (Register)
524	ECH	Exchange Accumulator with Storage
600	BOZ	Branch On Zero
601	BNZ	Branch On Non-Zero
604	BSN	Branch On Sense Unit
610	BOP	Branch On Positive
6104	BOM	Branch On Minus
700	BXH	Branch On Index High
710	BXL	Branch On Index Low
720	BXE	Branch On Index Equal
730	BSX	Branch and Set Index
740	BAX	Branch and Add to Index
750	BPX	Branch on Positive Index
760	BMX	Branch On Minus Index

In addition, the following addresses are available

$A	Accumulator
$L	Logical Register
$Z	Zero Cell
($A 1)	B Register

The user can add other instruction codes or addresses to LAP at any time by using CSET. For example:

```
CSET ($B 777622Q)
```

would define the address of the B register for LAP. (Actually $B is not in LISP at present, since the accumulator $A is at location 777621Q and $B can always be replaced by ($A 1) in LAP.)

Similarly:

```
CSET (AOR 53Q14)
```

could be used to define the instruction AOR (add one to register).

In writing LAP code, the programmer should be aware that it is the responsibility of each LAP procedure to save and restore the pushdown list and any registers it needs if it branches to some other procedure that can possibly induce the garbage collector.

1. A typical LAP function is shown in the following examples:

```
LAP (((ADD1 SUBR 1) (BAX ($ 2) 1 4) (Ø (E ADD1) 1)
      (BXH *PDLGN 1 *NDPDL) (STX -3 1 4) (STA 3 1)
      (LDA (QUOTE 1)) (BUC (*SPECIAL *PLUS) Ø 4) (BAX *RETRN 1 -4)) NIL)
```

2. Showing the use of LAP to correct a cell in a LISP program:

   Cell 4531ØQ is to be changed to do a branch on plus or minus zero to cell 4523Q (the number Ø is slashed)

   ```
   LAP ((4531ØQ (BOZ 45323Q Ø 3Q4)) NIL)
   ```

   The 34Q in the decrement changes the BOZ instruction from (left justified octal) 6ØØ to 6ØØ3 as required by the Q-32 to do a test on plus or minus zero. LAP returns NIL.

3. Use of LAP to insert a patch. The patch, to branch on zero accumulator to an error unwind is to be inserted at location 52345Q, which previously contained (STA 3 1). To insert a jump at 52345Q, one inputs LAP ((52345Q (BUC NIL)) NIL) LAP prints back NIL.

Then to complete the code, one inputs

```
LAP ((NIL (BOZ C ∅ 3Q4) (STA 3 1) (BUC 52346Q)
 C (LDA (QUOTE (ZERO ACCUMULATOR)))
   (BUC (*SPECIAL ERROR) ∅ 4)) NIL)
```

LAP returns ((C . 6∅∅54Q)).

If the patch was to be inserted in a vital portion of the LISP system, then it would be necessary to put it in reverse, so that the patch is operable before the jump. This is done as follows:

First insert

```
LAP (NIL A (BOZ C ∅ 3Q4)
           (STA 3 1) (BUC 52346Q)
         C (LDA (QUOTE (ZERO ACCUMULATOR)))
           (BUC (*SPECIAL ERROR) ∅ 4)) NIL)
```

LAP returns the locations of A and C ((A . 6∅∅51Q) (C . 6∅∅54Q)

Then insert the jump at 52345Q, knowing that the patch starts at 6∅∅51Q, using

```
LAP ((52345Q (BUC 6∅∅51Q)) NIL)
```

Note that one cannot use (BUC NIL) as in the previous method, since within LAP, the atom NIL stands for the current value of the binary program origin and the patch changed it.

Pushdown List:
In Q-32 LISP, the pushdown list is used to store pointers to arguments for functions and pointers to the values of program variables used by functions. It is also used to store return addresses for functions and to store the values of special variables which are used as LAMBDA variables or PROG variables, so that at the conclusion of operation of a function the previous value of a special variable can be restored. The index register 1 always contains a pointer to the top of the pushdown list at the level of either the current function or the previous function. Maintenance of the pushdown list is of paramount importance for all LISP functions. Q-32 LISP uses two machine cells for each entry on the pushdown list. The odd numbered cells store the current values, while even numbered cells are used to store previous bindings of special variables and are not directly used by the programmer.

When a function is entered, its arguments are always communicated as follows: Argument 1 is on the pushdown list at location 3 + PDP where PDP is the address pointer to the top of the pushdown list.

Argument 2 is on the pushdown list at location 5 + PDP.
Argument 3 is at 7 + PDP.
. . .
. . .
Argument i is at (2i+1) + PDP.

The last argument is in the accumulator.

Figure 4 shows the contents of the pushdown list at the time of entrance into a function of three arguments which uses two program variables. In the diagram, addresses increase upward. You will note that of the three arguments supplied to the function, the first and second arguments are pointed to from the pushdown list and the third and final argument is in the accumulator (hereafter this discussion will not distinguish between the value of an argument and a pointer to it).

If this function calls another function, which function can in turn possibly cause a garbage collection, it is the responsibility of the current function to do the following:

1. Protect its arguments by "bumping" PDP upwards N cells, where N = (number of arguments plus temporary variables + 2). This is done by the instruction,

   ```
   (BAX ($ 2) 1 N)
   ```

   which says to add to index 1 (PDP) the number N, which is the size of the block of cells on the pushdown list we wish to protect. Then go to the instruction at current location ($) plus 2. This BAX instruction must be present as the first instruction in all functions, even if N is zero.

2. The instruction at $+1 must always be of the form (∅ (E function-name) number-of-arguments number-of-pushdown-cells).

   This is not a real instruction, but is used in back tracing for error diagnostics.

3. Test to see whether the pushdown list is exhausted. This is done by the instruction

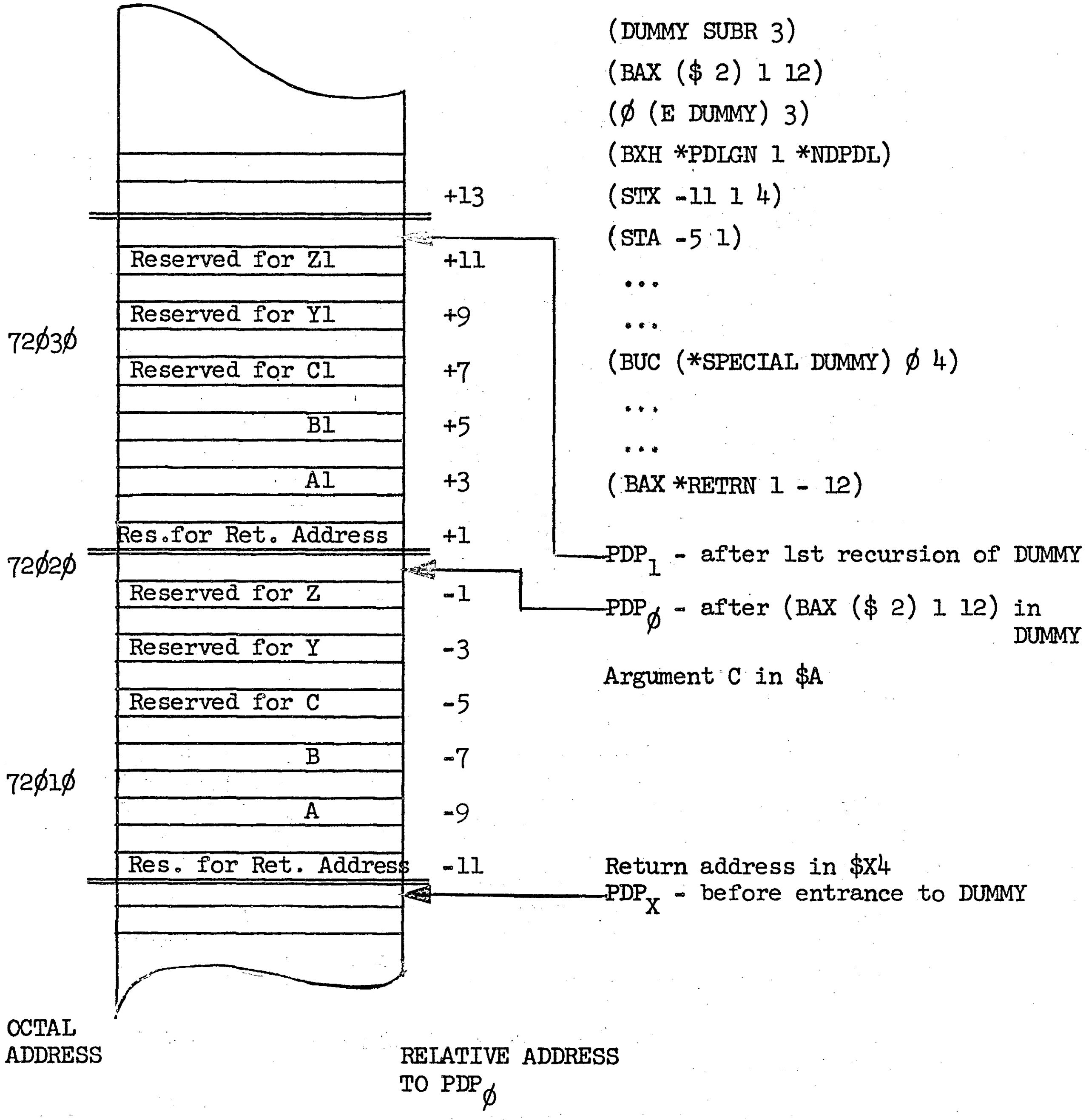
   ```
   (BXH *PDLGN 1 *NDPDL).
   ```

   This instruction is a conditional branch to an error routine (*PDLGN) if index 1 (PDP) as incremented by step 1 above is greater than the limit (*NDPDL) for the top of the pushdown list.

   Note that since we bumped the pushdown list pointer (PDP), the current function references the pushdown list for its arguments

At entrance to function:

```
(DUMMY (LAMBDA (A B C) (PROG (Y Z) (..... (DUMMY ..... ))))
```



```
(DUMMY SUBR 3)
(BAX ($ 2) 1 12)
(Ø (E DUMMY) 3)
(BXH *PDLGN 1 *NDPDL)
(STX -11 1 4)
(STA -5 1)
 ...
 ...
(BUC (*SPECIAL DUMMY) Ø 4)
 ...
 ...
( BAX *RETRN 1 - 12)
```

Fig. 4 Q-32 LISP Pushdown List

by, address = [(2) (argument number)+1]-N relative to the pushdown list pointer (PDP).

For the example in Fig. 4, N = 12, then
Argument 1 at ((2)(1)-12+PDP = -9+PDP
Argument 2 at -7+PDP
Argument 3 in the accumulator

4. Save index register 4 (RTN), which contains the return address of the calling routine, on the pushdown list at the beginning of the list of arguments. That is, treat the return address as another argument, argument Ø. Thus, it always is saved at location (1-n) + PDP. For the example in Fig. 4 the location is -11 + PDP. The instruction for this is

   (STX (1-N) 1 4).

5. Save the last argument which is in the accumulator on the pushdown list at the location appropriate to that argument number. For the example in Fig. 4, that argument is argument 3 and it would be saved at location

   ((2)(3)+1) -12+PDP = -5+PDP.

   (2+1)-4 = 3-4 = 1

   The instruction for this is

   (STA address 1)

   where "address" is as given in step 3 above.

6. If the current function calls another function, the current function must set up the arguments for that function. Arguments 1 through 4 for that function are entered at locations 3+PDP, 5+PDP, ... (2i+1)+PDP on the pushdown list, with the last argument entered in the accumulator.

7. Transfer control to the called function. This is done by the instruction,

   (BUC (*SPECIAL function-name) Ø 4)

8. When ready to return, load the accumulator with the value of the current function.

9. Reset the pushdown list pointer (PDP) to "unprotect" the arguments of this function just before exiting. This is done by the instruction,

   (BAX *RETRN 1 -N)

where N is as defined in step 1, and *RETRN is a closed subroutine for returning to the calling procedure.

There are some exceptions to this which are conveniences in writing LAP code. For example, if a function does not call another function then it need not adjust the pushdown pointer; also it need not store index 4, in which case the function does not need to execute steps 3 through 7, and the return, if index 4 has not been changed, is made via the instruction (BUC Ø 4) in lieu of step 9. However, instruction 1 must remain of the form (BAX ($ 2) 1).

Closed Subroutines:
A closed subroutine is written in LAP (it cannot be defined directly in LISP) by starting with (NAME SUBR Ø). Closed subroutines can be used wherever desirable. They usually have to include provision for storing the program counter into an instruction, and the last instruction in the subroutine is a BUC to whatever address was stored in that cell. There are several closed subroutines which are used by all arithmetic functions and which therefore deserve specific note. They are described here along with the special closed subroutine *LIST which is used by the compiler to expand the special form LIST.

*CØMPAT. This routine, starting with a pointer to a LISP number in the third cell of the pushdown list, and a pointer to a second number in the accumulator, returns with the value of the first in the accumulator and the value of the second in the B register, and index register 2 set to 2 if the final numbers are in floating point format, and set to 1 or 5 if they are in fixed point format. If either of the original LISP numbers pointed to was floating, the result is always floating.

*DIVIDE. *DIVIDE is a closed subroutine which uses *CØMPAT to place the numerical value of the first of two numerical arguments into the accumulator and the second in the B register, and to set index register 2 to 2 if the numbers are floating. *DIVIDE then divides the two numbers using either integer or floating point division, rounds a floating result, and returns with the quotient in the accumulator, the remainder in the B register, and index register 2 set to 1 for integer numbers or 2 for floating point numbers.

*FIXVAL. *FIXVAL is a closed subroutine which, starting with the accumulator pointing to a LISP number, exits with the integer part of the number in the accumulator.

*LIST. Is a subroutine which forms a list of n elements. It is used by the compiler in expanding the special form LIST. The calling sequence for *LIST is:

(BSX *LIST 2 n)
(Ø ℓ_1 1)
(Ø ℓ_2 1)
. . .
(Ø ℓ_n 1)

where n is the number of elements, and ℓ_1, ℓ_2 ... ℓ_n are pointers to the n elements to be listed.

*MKNO. This routine, given a number in the accumulator and the appropriate number 1, 2 or 5 in index register number 2, returns with the accumulator pointing to a LISP number of the appropriate representation.

*NUMVAL. This routine enters with the accumulator pointing to a LISP number and exits with the accumulator set to the same LISP number and with index register 2 set to the value 1 for a decimal integer, 2 for a floating point number and 5 for an octal number.

*PDLGN. *PDLGN is an error routine which prints out the message (OUT OF PUSHDOWN LIST) and unwinds LISP. As shown earlier in this section, this routine is called by most LISP functions by the instruction

```
(BXH *PDLGN 1 *NDPDL)
```

after the PDL pointer has been set.

*RETRN. *RETRN is not a closed subroutine, but is the common exit point for most of the system's closed subroutines and LAP functions. It consists of exactly two instructions,

```
(LDX 1 1 4)
```

and

```
(BUC Ø 4).
```

The first instruction loads index register 4 (RTN) from the contents of register 1+PDP. This is the location on the pushdown list of the return address of the current calling routine. It was set by the instruction

```
(STX (1-N) 1 4)
```

as we saw earlier in step 4 of section PUSHDOWN LIST. The second instruction branches to that return address.

The best way of determining how LAP programs are written is to trace LAP when defining a function or to perform CSET (PRINLIS T). The result will be the LAP code for that function which should be in accordance with the discussion herein.

4. RESERVED ATOMS IN Q-32 LISP

4.1 SUMMARY OF FUNCTIONS, FORMS, MACROS AND RESERVED ATOMS

This section summarizes the reserved atoms in Q-32 LISP as of 1 August 1965.

Table 1 lists all functions, special forms, atoms and macros which are of general utility to LISP users. These are described further in sections 4.2 through 4.9 as indicated in the table*.

Table 2 lists a set of functions which comprise the compiler and are not further described in this document.

Table 3 lists functions which constitute Evalquote in Q-32 LISP. They are described in section 4.3.

Table 4 lists reserved atoms, the inadvertent use of which can wreck the system.

Table 5 contains a list of character objects which are currently installed.

Section 4.2 describes those basic LISP functions which are essentially similar to those of 7090 LISP.

Section 4.4 describes general Q-32 LISP functions which are either not contained in 7090 LISP or which differ in some respects from 7090 LISP.

Section 4.5 describes arithmetic functions of Q-32 LISP, while section 4.6 describes buffer-handling functions. File input-output and library functions of Q-32 LISP are given in TM-2337/102/00 for Mod. 2.5 and TM-2337/111/00 for Mod. 2.6.

Several other functions have been described in earlier portions of the text: LAP and closed subroutines which are callable only from LAP code are described in section 3.6. MDEF and Macro were described in section 3.5.

*In Table 1 under Remarks, letters E and C denote functions used by Evalquote or the Compiler, respectively. Changing the definitions of any of these is likely to wreck the system.

Table 1. Available Functions

The following functions, special forms, atoms and macros are available in the system for general utility.

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
ABSVAL	SUBR	1	C	4.5
ADD1	SUBR	1	C	* 4.5
APPEND	SUBR	2	E C	4.4
AND	Form	Indef.	E C	* 4.2
ATOM	SUBR	1	C	* 4.4
BLANKS	SUBR	1	C	4.4
CAR	SUBR	1	E C	* 4.2
CDR	SUBR	1	E C	* 4.2
CAAR-CDDDDR	SUBR	1	E C	* 4.2
CHARP	SUBR	1	E C	4.4
COMPRESS	SUBR	1		4.4
*COMPAT	SUBR	Ø	L C	3.6
CONC	Form	Indef.	C	4.4
COND	Form	Indef.	E C	* 4.2
CONS	SUBR	2	E C	* 4.2
CSET	SUBR	2	E C	4.4
CSETQ	Form	2	E C	4.4
DEF1	SUBR	2	C	4.4
DEFILE	SUBR	1		4.6
DEFINE	SUBR	1	C	4.4

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote; C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 1 - Cont'd.

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
DEFLIST	SUBR	2	C	* 4.4
DELETEL	SUBR	2	C	4.4
DIFFERENCE	SUBR	2	C	* 4.5
DOTPAIR	SUBR	2		4.4
DIVIDE	SUBR	2		* 4.5
*DIVIDE	SUBR	∅	L	3.6
ENTIER	SUBR	1		4.5
EQ	SUBR	2	E C	* 4.2
*EQN	SUBR	2	- C	4.5
*EQP	SUBR	2	- C	4.5
EQUAL	SUBR	2	E C	* 4.4
*EQUAL	SUBR	3	- E C	4.4
EQUALN	SUBR	2	E C	4.4
ERROR	SUBR	1	E C	4.4
EVAL1	SUBR	1		4.4
EVALQT	SUBR	2	C	3.4
*EVALQT	SUBR	3	- E	4.3
EXP	SUBR	1		4.5
*EXPF	SUBR	1		4.5
EXPT	SUBR	2		4.5
*EXPTI	SUBR	2		4.5
*EXPAND	SUBR	2	E C	3.5
EXPLODE	SUBR	1		4.4

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote; C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 1 - Cont'd

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
F	Atom		E C	* 4.2
FIRST	SUBR	1		4.4
FIXP	SUBR	1	C	* 4.5
*FIXVAL	SUBR	∅	L C	3.6
FLOAT	SUBR	1		4.5
FLOATP	SUBR	1	C	* 4.5
FUNCTION	Form	1	E C	4.4
GENSYM	SUBR	∅	E C	* 4.4
GET	SUBR	2	E C	* 4.4
GETBUF	SUBR	2		4.6
*GETNO	SUBR	1	- C	4.4
GO	Form	1	E C	* 4.2
GREATERP	SUBR	2	E C	* 4.5
JUST	SUBR	1	C	4.5
LABEL	Form	2	E C	* 4.2
LAMBDA	Form	2	E C	* 4.2
LAP	SUBR	2	C	3.6
LAST	SUBR	1	C	4.4
LEFTSHIFT	SUBR	1	C	4.4
LENGTH	SUBR	1	E C	* 4.2
LESSP	SUBR	2	C	* 4.5
LIST	Form	Indef.	E C	* 4.2
*LIST	SUBR	∅	L C	3.6
*LOCN	SUBR	1	- C	4.4

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote, C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 1 - Cont'd

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
LOG	SUBR	1		4.5
LOGAND	Macro	Indef.		* 4.5
*LOGAND	SUBR	2	L	4.5
LOGOR	Macro	Indef.		* 4.5
*LOGOR	SUBR	2	L	4.5
LOGXOR	Macro	Indef.		* 4.5
*LOGXOR	SUBR	2	L	4.5
MACRO	SUBR	1		3.5
MAP	SUBR	2	C	* 4.4
MAPCAR	SUBR	2	E C	4.4
MAPCON	SUBR	2	C	* 4.4
MAPLIST	SUBR	2	C	4.4
MAX	Macro	Indef.		* 4.5
*MAX	SUBR	2	L	4.5
MEMBER	SUBR	2	E C	* 4.4
MIN	Macro	Indef.		* 4.5
*MIN	SUBR	2	L	4.5
MINUS	SUBR	1	C	* 4.5
MINUSP	SUBR	1	C	* 4.5
*MKNO	SUBR	∅	L E C	3.6
NCONC	SUBR	2	E C	* 4.2
NIL	Atom		E C	* 4.2
NOT	Form	1	E C	* 4.2
NULL	SUBR	1	E C	* 4.2
NUMBERP	SUBR	1	C	* 4.4

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote; C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 1 - Cont'd

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
*NUMVAL	SUBR	Ø	L C	3.6
OBLIST	Atom	Ø	E C	* 4.2
OR	Form	Indef.	E C	* 4.2
PAIR	SUBR	2	C	* 4.2
*PDLGN	SUBR	Ø	L E C	3.6
*PLANT	SUBR	2	- C	4.4
PLUS	Macro	Indef.		* 4.5
*PLUS	SUBR	2	L	4.5
PRINØ	SUBR	1	E C	4.4
PRIN1	SUBR	1	E C	* 4.4
PRINT	SUBR	1	E	* 4.4
PRINTCH	SUBR	1		4.4
PROG	Form	Indef.	E C	* 4.2
PROG2	SUBR	2	E C	* 4.2
PROP	SUBR	3	C	* 4.4
QUOTE	Form	1	E C	* 4.2
QUOTIENT	SUBR	2		* 4.5
*RATOM	SUBR	Ø	E	4.4
READ	SUBR	Ø	E	* 4.4
READ1	SUBR	Ø	E	4.4
READCH	SUBR	Ø		4.4
REMAINDER	SUBR	2		* 4.5
*RETRN	SUBR	Ø	L C	3.6
RETURN	Form	1	E C	* 4.2
REVERSE	SUBR	1	C	* 4.2
RPLACA	SUBR	2	E C	* 4.2

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote; C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 1 - Cont'd

Name	Type[1]	No. of Arguments	Remarks[2]	Section[3]
RPLACD	SUBR	2	E C	* 4.2
SASSOC	SUBR	3	C	* 4.2
SAVE	SUBR	1		4.6
SELECT	Form	Indef.	E C	* 4.2
SET	Form	2	not implemented	4.4
SETBUF	SUBR	2		4.6
SETQ	Form	2	E C	* 4.4
SPECIAL	SUBR	1	E	4.4
*SPECBIND	SUBR	1	L C	3.6
SQRT	SUBR	1		4.5
SQUOZE	SUBR	1		4.4
SUB1	SUBR	1		* 4.5
SUBST	SUBR	3	C	* 4.4
T	Atom		E C	* 4.2
TEREAD	SUBR	Ø	E	4.4
TERPRI	SUBR	Ø	E	* 4.4
TIMES	Macro	Indef.		* 4.5
*TIMES	SUBR	2	L	4.5
TRACE	SUBR	1		* 4.4
UNSPECIAL	SUBR	1	E	4.4
UNTRACE	SUBR	1		* 4.6
ZEROP	SUBR	1	C	* 4.5

[1] SUBR = function; Form = special form; Atom = special atom

[2] - = for system programming only, not general programming; L = useful within LAP code only; E = used by Evalquote; C = used by Compiler

[3] Section in which described; * = same as 7090 LISP

Table 2. Compiler Functions

The following set of functions are used by the Compiler, and are not directly useful to most LISP users:

- ATTACH
- *CØ496
- *CØ586
- *CØ6Ø1
- *CØ895
- *C17Ø6
- *C1852
- *C1927
- *C1946
- *C217Ø
- *C2293
- *C2572
- *C2598
- *C2671
- *C2854
- *C2859
- CALL
- CEQ
- *CLRFLAG
- COMBOOL
- COMCOND
- COMLIS
- COMP
- COMPACT
- COMPLY
- COMPROG
- COMVAL
- *CONDERR
- DIFFER
- LAC
- LAPEVAL
- LOCATE
- LSHIFT
- MDEF
- *MKTRC
- *MKUNT
- PA1
- PA3
- PA4
- PA5
- PA7
- PA8
- PA9
- PA12
- PAFORM
- PAIRMAP
- PALAM
- PASS1
- PASS2
- P11
- P13
- PROGITER
- *SETFLAG
- SPECBIND
- SPECRSTR
- STORE

Table 3. Evalquote Functions

The following seven functions constitute Evalquote in Q-32 LISP and are not directly interesting to most LISP users:

COM2

*DEFQ

*EVQ

*EVALQT

*EXPQ

*MGSYM

*SUPV

Table 4. Reserved Atoms

Other than the names of functions, special forms, macros, and atoms used with LAP as noted herein, there exists a collection of ATOMS reserved by the system for Evalquote, LAP, and the computer, which should be avoided by the user at all cost. Their unintentional use could wreck the system. These reserved ATOMS include:

*NIL	-	bound to location ∅
BPORG	-	pointer to next cell for compilation
*BPORG	-	backup pointer for BPORG
*SCRACH	-	pointer to start of Scratch area
TBPS	-	pointer to top of Binary Program Space
PRINLIS	-	used as free variable by compiler
*SCRACHX	-	pointer to next cell in Scratch
*NDPDL	-	constant used to test for top of Pushdown List

The following atoms are clobbered by Evalquote and cannot be bound by the user:

*FGNSL *FVAL *VALIST

Table 5. Character Objects

The system currently contains an incomplete set of Hollerith objects.

The atom names below are bound to character values which print as indicated. Of course, any character on the teletype can be entered into the Q-32 system via the $$ artifact which is available as specified in the LISP 1.5 Manual for 7090 LISP.

Object	Value	Object	Value
LPAR	(	COLON	:
PERIOD	.	LARR	←
BLANK	Blank	UPARR	↑
RPAR	)	LSTHAN	<
DOLLAR	$	GRTHAN	>
STAR	*		
SLASH	/		
EQSIGN	=		

Any other Hollerith objects can be added by the user as desired, viz., CSET (PERCENT $$/%/)

4.2 COMMON LISP FUNCTIONS

This section contains brief descriptions of atoms and functions which are common to Q-32 LISP and 7090 LISP and which act the same in both. Their names are:

AND
CAR
CDR
CAAR - CDDDDR
COND
CONS
EQ
F
GO
LABEL
LAMBDA
LENGTH
LIST
NCONC
NIL
NOT
NULL
OBLIST
OR
PAIR
PROG
PROG2
QUOTE
RETURN
REVERSE
RPLACA
RPLACD
SASSOC
SELECT
T

In addition, the following functions are available in Q-32 LISP but are slightly different from the same functions in 7090 LISP. They are described in section 4.4.

APPEND
ATOM
CONC
CSET
CSETQ
DEFINE
DEFLIST
EQUAL
GENSYM
GET
LEFTSHIFT
MAP
MAPCON
MAPLIST
MEMBER
PRIN1
PRINT
PROP
READ
SETQ
SPECIAL
SUBST
TERPRI
TRACE
UNSPECIAL
UNTRACE

Function Descriptions:
In the following description, all functions and forms which can be given to Evalquote at the top level are shown in external Evalquote form. Those which cannot be given to Evalquote directly are shown in internal LISP format.

AND (x_1, x_2 ... x_n) Special Form Predicate (Compiler, Evalquote)

AND is a special form of an indefinite number of arguments. Its arguments are evaluated in succession until one of them is found to be NIL (false) or until the end of the list is reached. The value of AND is then NIL or T (true) respectively. The value of (AND) of no arguments is T (true).

CAR (x) SUBR (Compiler, Evalquote)

If x, the argument of CAR, points to a character atom or NIL, CAR induces the error ((CAR CHARACTER) NOT DEFINED) or ((CAR NIL) NOT DEFINED). Otherwise, CAR leaves in the accumulator the entire word which was pointed at by the address portion of x. (This makes it possible to use CAR to transmit an entire instruction word, as required for example by the function *EVQ. Almost all LISP functions look at only the address portion of the accumulator, and so the effect of CAR is to return the address portion of x).

CDR (x) SUBR (Compiler, Evalquote)

If x, the argument of CDR, points to a character atom or NIL, CDR induces the error ((CDR CHARACTER) NOT DEFINED) or ((CDR NIL) NOT DEFINED). Otherwise, CDR returns the contents of the decrement of x.

CAAR-CDDDDR are all defined in Q-32 LISP as composition functions of CAR and CDR.

(COND (p_1 e_1) (p_2 e_2) ... ($p_n e_n$)) Special Form (Compiler, Evalquote)

The special form COND takes an indefinite number of argument clauses in the form of pairs (p_i e_1). where p_i is a predicate and e_i is a form.

The parentheses in COND have a different meaning than they do in any other LISP form, in that (p_1 e_1) does not mean to apply function p_1 to argument e_1. Instead, p's are evaluated from left to right until the first one, say p_t, is found that evaluates to true, or specifically, is not EQ to NIL. The effect of the entire COND is that of the associated form e_t ; all other e_i $(i \neq t)$ and p_i $(i > t)$ are not evaluated (operated).

p_i may in general be any form in LISP except the specific statement forms (GO label) or (RETURN value) since these have no value.

If COND is used anywhere except as a top level statement within PROG, then at least one of the p_i must be true (typically, the last p_n is the atom T). If none are true, an error will be detected at operate time. (If none of the p_i are the atom T, the compiler inserts the pair (T (*CONDERR)) at the end of the COND.) None of the expressions e_i may be of the form (GO label) or (RETURN value).

If COND is used at the top level of a PROG, then it is a statement executed for effect not value (except when an e_i is of the form (RETURN value)),and the following differences occur:

1. The requirement that one of the p_i be true is waived. If all p_i are NIL, the COND falls through to the next statement.

2. Any of the forms e_i may be of the form (RETURN $\underline{v}$), where $\underline{v}$ is an expression to be evaluated and is the value of the PROG.

3. Any of the forms e_i may be a statement of the form (GO(ℓ) where $\underline{\ell}$ must be a label which is used in this PROG (see PROG).

Because of the peculiar method of evaluating its arguments, COND cannot be used as the function name at the top level of Evalquote.

CONS ($\underline{x}$ $\underline{y}$) SUBR (Compiler, Evalquote)

CONS is a basic function of LISP which takes a cell from free storage, places $\underline{x}$ and $\underline{y}$ in the address portions and decrement portions of the cell, respectively, and returns a pointer to the address of the new cell.

EQ ($\underline{x}$ $\underline{y}$) SUBR (Special Form) (Compiler, Evalquote)

EQ tests for equality of the addresses of its two arguments $\underline{x}$ and $\underline{y}$. If $\underline{x}$ and $\underline{y}$ are the same literal atom, the result is T. If both arguments are numbers or lists, then EQ is undefined. In all other cases, EQ returns NIL.

When it is used as a predicate, EQ is compiled as open code. For other uses, the function EQ is defined by:

(EQ (LAMBDA (A B) (COND ((EQ A B) T) (T NIL))))

F Special Atom (Compiler, Evalquote)

The Special Atom F may be regarded as permanently evaluated to NIL. Whenever F is encountered outside of a quoted expression, the compiler replaces F with (QUOTE NIL), whose value is later changed to NIL (address = ∅) by LAP. Thus, F cannot be bound by LAMBDA or PROG. However, F does not yield NIL until it is evaluated, and is not the same as NIL when given to Evalquote, which quotes its arguments. Thus:

AND (T NIL) = NIL while

AND (T F) = T,

since F in quoted status is not NIL, and thus is true.

On the other hand,

(LAMBDA ( ) (AND T F)) ( ) = NIL,

for in this expression F is evaluated, not quoted, but

(LAMBDA ( ) (AND (QUOTE T) (QUOTE F))) ( ) = T.

The atom F can be given a zero level binding by CSET, e.g., CSET (F FALSE), but the binding is not useful, since it can be picked up only by performing (CAAR (QUOTE F)).

(GO ℓ) Special Form PROG only (Compiler, Evalquote)

GO is a special form valid only within PROG. (GO ℓ) causes the flow of the program to move to the label ℓ within the PROG. See PROG.

(LABEL name (LAMBDA-expression)) Special Form

LABEL is a special form used to give a LAMBDA-expression a name so that it can be called recursively from within the LAMBDA-expression. LABEL causes compilation to occur in a manner similar to DEFINE in Q-32 LISP, but with two differences:

1. the name used in LABEL is local, and can be seen only within this LABEL form. It thus can cause no conflict with other functions or atoms used in the system.

2. LABEL defines a single function, while DEFINE can take many functional expressions.

(LAMBDA args expr) Special Form (Compiler, Evalquote)

A LAMBDA-expression defines a LISP function, in terms of a list containing the atom LAMBDA; args is a list of atoms (a_1 a_2 ... a_n) (or the empty list ( ) or NIL); expr is any single form (S-expression).

LAMBDA serves several functions. First it is a flag telling LISP that the next list is to be interpreted as a list of arguments rather than a function to be evaluated. Second, the S-expression which follows the argument list tells LISP how to evaluate the LAMBDA expression and compute its value. In Q-32 LISP, LAMBDA-expressions are always compiled into functions, and evaluation consists of operating the compiled code.

The atoms a_i, in args must be distinct literal atoms, not including F, NIL, T; and if an atom is to have a functional binding, it must not be the same as any Special Form (for clarity, the use of the name of any function or Macro should be avoided).

The args a_i are in general dummy arguments for the LAMBDA expression or function, and refer only to cells on the pushdown list. However, if any of the a_i have been declared Special prior to compilation, then the actual atom a_i is used in the function and the compiled code saves the prior binding of a_i on the pushdown list, binds the atom a_i temporarily during the operation of the function, and restores the previous binding when exiting from the function.

The S-expression is any single function or PROG to be evaluated, using some or all of the arguments a_i. If the LAMBDA-expression is used within DEFINE, MACRO or LABEL and has a name, this name may be used in its own definition.

A LAMBDA-expression is syntactically equivalent to a function name and may be freely used wherever a function name is legal. For example, the following expressions are completely interchangeable forms of the function CDR

```
  i) CDR

 ii) (LAMBDA (X) (CDR X))

iii) ((LAMBDA (Y) Y) (FUNCTION CDR))

 iv) (LAMBDA (X) (((LAMBDA (Y) Y) (FUNCTION CDR)) X))

  v) ((LAMBDA (Y) Y) (FUNCTION (LAMBDA (X) (CDR X))))
```

in the above, iv) was obtained by substituting iii) into ii), while v) was obtained by substituting ii) into iii).

Also, the following expressions operate identically:

```
((LAMBDA (G) (LIST G G G)) (GENSYM))
```

and

```
(PROG (G) (SETQ G (GENSYM)) (RETURN (LIST G G G)))
```

LENGTH (x) SUBR (Compiler, Evalquote)

LENGTH applied to a list x returns an integer equal to the number of elements in the top level of the list. Applied to an atom it yields zero.

```
(LENGTH (LAMBDA (M) (PROG (N) (SETQ N Ø)
 A (COND ((ATOM M) (RETURN N)))
 (SETQ M (CDR M)) (SETQ N (ADD1 N)) (GO A))))
```

LIST (x_1 x_2 ... x_n) Special Form (Compiler, Evalquote)

LIST takes an arbitrary number of arguments, and constructs a list out of them. The compiler handles the Special Form LIST by constructing open code using the function *LIST (see section 3.6).

*LIST calls CONS and the effect is the same as

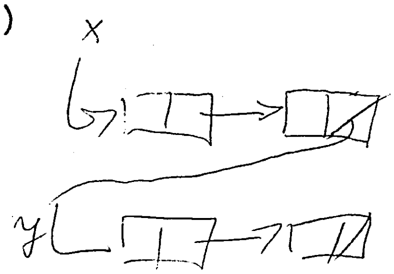
(CONS X_1 (CONS X_2 (... (CONS X_n NIL) ...)))

but the actual method employed in Q-32 LISP is considerably more efficient in terms of length of compiled code and speed of operation if n > 2.

NCONC (x y) SUBR (Compiler, Evalquote)

NCONC appends list y onto the end of list x, without copying x. The value of NCONC is the new value of x. The NULL test is used to find the end of the list x. If x is atomic, NCONC appends a y onto the end of the property list of atom x.

```
(NCONC (LAMBDA (X Y) (PROG (M)
  (COND ((NULL X) (RETURN Y))) (SETQ M X)
 A (COND ((NULL (CDR M))(GO B)))
  (SETQ M (CDR M))(GO A)
 B (RPLACD M Y) (RETURN X))))
```



NIL Special Atom (Compiler, Evalquote)

NIL is equivalent to the empty list ( ) and is treated by the compiler as a pointer to address zero. On input, ( ) is read as NIL; in the Compiler, NIL is converted to (QUOTE NIL) while (QUOTE NIL) is unchanged (see QUOTE). Thus, ( ), NIL and (QUOTE NIL) all arrive at LAP as (QUOTE NIL) and LAP replaces (QUOTE NIL) by address $Z or 7776Q2 which contains zero.

The atom NIL actually exists in the system but is used only for reading and writing the print name NIL, and is not accessible for binding.

NOT (x) Special Form (Compiler, Evalquote)

NOT is regarded as a Special Form by the Compiler, and (NOT X) is always changed to the equivalent form (NULL X).

NULL (x) SUBR

NULL is compiled as open code when used as a predicate. For other uses, the definition used is

```
(NULL (LAMBDA (X) (COND ((NULL X) T) (T NIL))))
```

OBLIST Special Atom (Compiler)

The atom OBLIST has a zero-level binding to a list of 125 buckets which occupy adjacent cells in core. From the Ith bucket (I = Ø, 1 ... 124) are strung all literal atoms for which the remainder is I when the absolute value of the first word of the print name (treated as a number) is divided by 125.

The OBLIST in Q-32 LISP is used primarily as a dictionary for reading literal atoms, and does not always protect atoms. If atom head space is exhausted, marking of atoms for protection from garbage collection is done from the pushdown list, quote cells, and atom head space only. Atoms which have no property list are not pointed to and have no binding are removed, and the remaining literal atoms and gensyms are bucket sorted and restrung to form a new OBLIST.

OR (p_1 p_2 ... p_n) Special Form (Compiler, Evalquote)

The arguments of OR are evaluated from left to right until the first true (non-NIL) predicate is found. If a true predicate is found, the value of OR is T; if the end of the list is reached, the value of OR is NIL. The value of (OR) of no arguments is NIL.

PAIR ($\underline{x}$ $\underline{y}$) SUBR (Compiler)

PAIR requires its inputs $\underline{x}$ and $\underline{y}$ to be lists of equal length

$\underline{x} = (x_1\ x_2\ \ldots\ x_n)$ $\underline{y} = (y_1\ y_2\ \ldots\ y_n)$.

PAIR returns a list of dotted pairs

$((x_1 \cdot y_1)\ (x_2 \cdot y_2)\ \ldots\ (x_n \cdot y_n)$
as its value if this condition is met.

If the two lists are of unequal length, PAIR induces the ERROR returns

((PAIR ERROR F2) $\underline{x}$ $\underline{y}$) if $\underline{x}$ is shorter than $\underline{y}$

or ((PAIR ERROR F3) $\underline{x}$ $\underline{y}$) if $\underline{y}$ is shorter than $\underline{x}$.

(PROG $\underline{\text{vars}}$ s_1 s_2 ... s_n) Special Form (Compiler, Evalquote)

PROG is a Special Form that permits LISP programs to be written in the form of a series of statements to be executed. In form, PROG looks like a function of an indefinite number of agreements.

Its first argument $\underline{\text{vars}}$ must be either an empty list or a list of atomic symbols $(v_1\ v_2\ \ldots\ v_n)$, called $\underline{\text{program variables}}$. Any program variable which is not in Special status at compile time is merely a cell on the pushdown list. If a program variable is in Special status during compilation, its previous binding is saved on the pushdown list at entrance to the PROG and is restored at exit, and the current binding is stored in the CAR of the atom head within the PROG. Thus in either case, the binding of a program variable is visible only within the PROG. However, if the variable is Special, it is also visible when used free by any function called from within the PROG. If not Special, it is invisible except in the body of the PROG.

The other arguments s_1 ... s_n of a PROG can be either atoms or statements. An atom is regarded simply as a label which is local to the PROG. A statement may be any standard LISP form or expression or may include a GO statement or a RETURN statement. If there are no GO or RETURN statements, the statements s_1 s_2 ... s_n are executed by evaluating the corresponding LISP form and ignoring the value. (Atoms are disregarded since evaluating an atom and discarding the value is of no consequence.) The control "falls out" of the PROG at the end, and the value of the PROG is NIL.

The form (GO ℓ), where ℓ is a label within the PROG, can occur at the top level of the PROG as one of the s_i or can be used at the top level of a COND or SELECT at the top level of the PROG. If evaluated (GO ℓ) causes transfer of control to the label ℓ in the PROG.

The form (RETURN v) can occur under the same conditions as (GO ℓ) but causes v to be evaluated, and causes exit from the PROG, with v as the value of the PROG.

Within a PROG, COND does not require a T alternative, since control simply "falls through" to the next statement. SELECT with NIL as its final expression causes the same effect.

PROG2 (a b) SUBR (Compiler, Evalquote)

PROG2 causes its first argument to be evaluated and returns the value of its second argument. It is equivalent in result to (PROG ( ) a (RETURN b)). It is defined by (PROG2 (LAMBDA (X Y) Y))

(QUOTE v) Special Form (Compiler, Evalquote)

The value of the special form QUOTE is the CADR of the list whose CAR is the atom QUOTE. Thus, when evaluated,

(QUOTE A) = A

(QUOTE (A B)) = (A B), etc., but

(QUOTE A B) = (QUOTE A . B) = A

A quoted expression stands for itself, and is not evaluated.

In Q-32 LISP structure the form (QUOTE A) is represented by a quote cell which points to atom A; similarly the form (QUOTE (A B)) is a quote cell which points to the list (A B). Hence, when (QUOTE A) or (QUOTE (A B)) is transmitted to a function, it is the address of the corresponding quote cell which points to the desired LISP object or list.

Constants which are numbers, character atoms, T, F and NIL need not be quoted in Q-32 LISP because the compiler always replaces the constant n by the form (QUOTE n), and the quote cell (QUOTE n) is a pointer to the constant n. (F becomes (QUOTE NIL))

(RETURN exp) Special Form (Compiler, Evalquote)

The Special Form (RETURN exp) is legal only at the top level of a PROG or at the top level of a COND within a PROG. If (RETURN exp) is encountered in evaluation of a PROG, the expression exp is evaluated (operated) and its value is the value of the PROG.

REVERSE (ℓ) SUBR (Compiler)

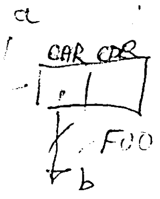
The function REVERSE has for its value a list whose elements are the top level elements of list ℓ taken in reverse order, e.g.,

REVERSE ((A (B C) D (E F))) = ((E F) D (B C) A)

When applied to an atom or to a list terminated by an atom other than NIL, REVERSE is undefined.

RPLACA (a b) SUBR (Compiler, Evalquote)

RPLACA replaces the CAR of the cell pointed to by a with the pointer b. Its value is a but a has been replaced by (CONS b (CDR a))



For example, RPLACA (PI NIL) would unbind a previous CSET value of PI. CSET (PI 3.14159) is equivalent to

RPLACA (PI (3.14159))

SPECIAL ((PI))

RPLACD (a b) SUBR (Compiler, Evalquote)

RPLACD replaces the CDR of the cell pointed to by a with the pointer b. Its value is a but a has been replaced in value by (CONS (CAR a) b).

The use of RPLACD on an atom will destroy the print name of the atom and can easily wreck the system. In fact, no useful result can occur from the use of RPLACD at the top level of Evalquote.

(SASSOC x y fn) SUBR functional (Compiler)

SASSOC searches y which is a list of pairs (usually but not necessarily dotted pairs), for the first pair whose first element is EQ to x. If the search succeeds, the value of SASSOC is the pair. If the search fails, the value of SASSOC is (fn), a function of no arguments.

Because of its functional argument, SASSOC cannot be input as a function at the top level of Evalquote.

```
(SASSOC (LAMBDA (X Y FN) (PROG ( ) A (COND ((NULL Y) (RETURN (FN)))
 ((EQ (CAAR Y) X) (RETURN (CAR Y))))
  (SETQ Y (CDR Y)) (GO A))))
```

(SELECT a_o (a_1 e_1) (a_2 e_2) ... (a_n e_n) e_o)

Special Form (Compiler, Evalquote)

The expression a_o is evaluated, then each of the a_i are evaluated in turn and tested until the first one is found that satisfied (EQ a_o a_i). The value of SELECT is then the corresponding e_i. If no such a_i is found, the value of SELECT is e_o.

SELECT can be used at the top level of PROG in much the same way as COND. In this application GO and RETURN forms are legal for e_i and e_o. However, e_o cannot be omitted, but may be NIL.

The compiler converts SELECT to the equivalent form

((LAMBDA (G) (COND ((EQ G a_1) e_1) ((EQ G a_2) e_2) ...
((EQ G a_n) e_n) (T e_o))) a_o)

where G is an arbitrary gensym. (If e_o were omitted, the syntax of the COND would be incorrect.)

T Special Atom (Compiler, Evalquote)

The special atom is permanently bound to the value T. Whenever T is encountered outside a quoted expression, the compiler replaces T by (QUOTE T). Thus T cannot be bound by LAMBDA or PROG. It may be bound at zero level by CSET (T TRUE), for example, but the binding cannot be picked up except by explicitly performing (CAAR (QUOTE T)) or (CAAR T), and so is not normally of any use.

4.3 EVALQUOTE FUNCTIONS

The six functions described in this section constitute the Q-32 LISP Evalquote. The function COM2 is a principal function of the compiler in addition to being used by Evalquote. These functions are not of use to most LISP users.

*SUPV ()

The supervisor *SUPV is a function of no arguments which calls for two S-expressions to be read from the teletype, terminates the input buffer, then calls for (PRINT (*EVALQT X Y (QUOTE *FUNC))) and loops back to call for two more S-expressions. (Here X and Y are the two S-expressions read.)

*EVALQT (fcn args name)

*EVALQT evaluates fcn as a function with arguments listed in args as follows:

If fcn is an atom and is a true function (not a macro or special form), then *EVALQT calls *EVQ to operate the function;

in all other cases, an appropriate Lambda-expression is manufactured and compiled under the name name into a reusable scratch area of core, using functions *DEFQ, *MSGYM, MDEF, COM2.

Then *EVALQT calls *EVQ to operate the function and returns the value of this function applied to its arguments.

*EVQ (locn args)

*EVQ operates the function whose code starts at locn with args as its list of arguments, and returns the value of the function. Note that if fcn is a function name, then locn is in general (CAAR fcn).

*DEFQ (name fcn args)

*DEFQ is a defining function which is used by *EVALQT to prevent temporarily compiled functions from using up quote cells or giving permanent bindings to gensyms. It uses *MGSYM, MDEF and COM2.

*MGSYM (value)

*MGSYM is a macro expander used by *DEFQ to remove numbers and quoted quantities from an expression before compilation, and replace them by additional arguments. The removed values are stored on *VALIST. (*EVQ (CAAR FCN) (APPEND (GET *VALIST FCN) ARGS))

COM2 (type nargs exp name)

COM2 is the function which is used by the compiler, DEFINE, MACRO and *DEFQ to do the final compilation of all functions. Type is always SUBR. Nargs is the number of arguments which the function expects. Exp is the LAMBDA expression for the function. Name is the name of the function. COM2 cannot handle Macros, hence MDEF must be applied to Exp before COM2 is called.

COM2 is called in the compiler by COMP to compile all LABEL expressions and all LAMBDA expressions used as functional arguments after FUNCTION.

4.4 Q-32 GENERAL PURPOSE LISP FUNCTIONS

This section describes LISP functions which are either different from 7090 LISP functions of the same name or are entirely new functions of general utility. The names are listed below. The sign # before the name indicates an entirely new function.

	APPEND
	ATOM
#	BLANKS
#	CHARP
#	COMPRESS
	CONC
	CSET
	CSETQ
	DEF1
	DEFINE
	DEFLIST
#	DELETEL
#	DOTPAIR
	EQUAL
#	*EQUAL
#	EQUALN
	ERROR
#	EVAL1
#	EXPLODE
#	FIRST
	FUNCTION
	GENSYM
	GET
#	*GETNO
	LAST
	LEFTSHIFT
#	*LOCN
	MAP
#	MAPCAR
	MAPCON
	MAPLIST
	MEMBER
	NUMBERP
#	*PLANT
#	PRINØ
	PRIN1
	PRINT
#	PRINTCH
	PROP
#	*RATOM
	READ
#	READ1
#	READCH
	SETQ
	SPECIAL
#	SQUOZE
	SUBST
#	TEREAD
	TERPRI
	TRACE
	UNSPECIAL
	UNTRACE

APPEND (x y) SUBR (Compiler, Evalquote)

If x is not an atom, APPEND returns a copy of x in which y replaces the CDR of the last cell at the top level. If x is an atom, APPEND is undefined.

Examples:

1) APPEND ((A B) (C D)) = (A B C D)
2) APPEND ((A B) C) = (A B . C)
3) APPEND ((A . B) C) = (A . C)

For both arguments in the form of lists (Example 1), the result is the same as in 7090 LISP. The other cases are undefined and cause errors in 7090 LISP.

ATOM (x) SUBR Predicate (Compiler, Evalquote)

Atom returns T (true) if x is any atom, and NIL otherwise. ATOM is true for all atoms, including numbers.

BLANKS (n) SUBR (Compiler)

BLANKS (n) enters n blanks into the output buffer used by PRINT. If n is not a number, an error will result. If n is not a positive integer, an endless loop will result.

CHARP (c) SUBR (Compiler, Evalquote)

CHARP is a predicate that tests for character atoms. The value of CHARP is T if c is a character atom (address in the range 10000Q to 10077Q) and NIL if c is not a character atom.

COMPRESS (ℓ) SUBR

COMPRESS is a LISP function that returns as its value a "literal" atom formed from the list of character atoms ℓ. If ℓ is a character atom, COMPRESS returns the (SQUOZE (LIST ℓ)). If ℓ is any other atom, COMPRESS returns ℓ. If ℓ is a list of anything other than character atoms, COMPRESS returns an error message and "unwinds."

CONC (x_1 x_2 x_3 ... x_n)

CONC acts like an APPEND of many arguments and concatenates its arguments onto one new list. The first argument is copied. (The compiler actually treats CONC by performing a Macro expansion in terms of APPEND.)

CSET (a v) SUBR (Compiler, Evalquote)

This function is most useful at the top level of Evalquote. It is used to establish a zero-level binding of an atom.

CSET (PI 4.13159) sets the value of the atom PI to the value 3.14159. (Note that both arguments of CSET are quoted by Evalquote.

The form (CSET a v) produces the following results: If a is not an atom (i.e., does not have a quoted atom as its value) an error is detected; otherwise a is made Special, and then the effect is the same as (RPLACA (CAR a) (LIST v))

(CSETQ a v) Special Form (Compiler, Evalquote)

This Special Form is like CSET except that it quotes its first argument, which must be an atom. CSETQ cannot be used at the top level of Evalquote.

DEF1 (ob ℓ) SUBR (Compiler)

DEF1 is a subsidiary function used by DEFLIST to place the value ℓ on the property list of the object ob under the property named PRO. PRO is a free variable which must be set previously.

DEFINE (x) SUBR

The argument of DEFINE, x is a list of pairs

$((n_1\ d_1)\ \ (n_2\ d_2)\ \ldots\ (n_n\ d_n))$,

where each n_i is a name of a function and d_i is the corresponding LAMBDA-expression for the function.

The execution of DEFINE is as follows:

Each pair (n d) is compiled, and a pointer to the compiled code for the function n is placed in the CAR of the atom n; the expressions d are discarded and DEFINE returns a list of the n's.

If any error occurs in compilation, the definition in which it occurs, and all subsequent definitions, are not compiled, but any previous compilations are unaffected.

If DEFINE is used twice on the same atom, the new definition replaces the old, and the old binary program space in general is lost.

DEFLIST (x ind) SUBR (Compiler)

The first argument of DEFLIST x is a list of pairs ((n_1 d_1) (n_2 d_2) ...), as for DEFINE, and the second argument ind is an atom. DEFLIST places each expression d on the property list of the corresponding atom n under the indicator ind. (Note that there is in Q-32 LISP no relation between DEFINE and DEFLIST.)

DEFLIST is used by the function MACRO.

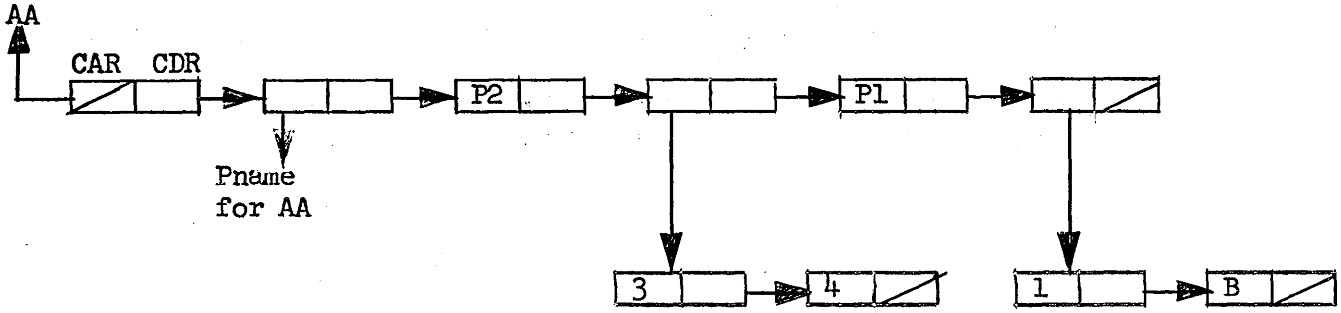
If DEFLIST is used twice on the same atom with the same indicator, the old expression on the property list is replaced by the new one. DEFLIST places new properties on the property list to the left of all old properties.

For example:

DEFLIST(((AA (1 B)))P1)

DEFLIST(((AA (3 4)))P2)

results in the following structure for atom AA



DELETEL (b m) SUBR (Compiler)

DELETEL deletes from list m all elements which are members of list b, and reCONSes the remaining elements into a new list. It does not change m. (DELETEL (LAMBDA (B M) (MAPCON M (FUNCTION (LAMBDA (J) (COND ((MEMBER J B) NIL) (T (LIST J))))))))

It returns the new list as its value.

DOTPAIR (a) SUBR Predicate

DOTPAIR is a predicate that is true if a is atomic or a dotted pair of atoms, and is false otherwise. It is used by the function FIRST to find the first printable object on a list.

EQUAL (x y) SUBR (Compiler)

EQUAL tests x and y for equality by going down the CAR and CDR chains and using the function EQ to test equality of literal atoms and the function *EQP to test numbers (see section 4.5).

EQUAL is defined by

```
  (EQUAL (LAMBDA (X Y) (*EQUAL X Y (FUNCTION *EQP))))
```

(*EQUAL x y fn) SUBR (Compiler, Evalquote)

*EQUAL tests x and y for equality recursively going down both CAR and CDR chains. Numbers are compared using function fn.

```
(*EQUAL (LAMBDA (A B FN) (COND ((EQ A B) T
((NUMBERP A) (COND ((NUMBERP B) (FN A B)) (T F)))
((ATOM A) F) ((ATOM B) F)
((*EQUAL (CAR A) (CAR B) FN) (*EQUAL (CDR A) (CDR B) FN)) (T F))))
```

EQUALN (x y) SUBR Predicate (Compiler, Evalquote)

EQUALN tests whether two lists x and y are identical. It uses *EQN to test numbers and will fail if two numbers are unequal in value or differ in representation. It is used to test whether quoted constants are identical, and is also used by SUBST.

```
(EQUALN (LAMBDA (X Y) (*EQUAL X Y (FUNCTION *EQN))))
```

ERROR (msg) SUBR (Compiler, Evalquote)

ERROR causes its arguments msg to be evaluated and its value printed and then induces an error unwind of the LISP system.

EVAL1 (exp) SUBR

EVAL1 performs evaluation of one S-expression exp. It is defined by (EVAL1 (LAMBDA (S) (EVALQT (LIST (QUOTE LAMBDA) NIL S) NIL)))

EXPLODE (a) SUBR

EXPLODE is a LISP function that returns as its value a list of character atoms "exploded" from the print name of atom a. If a is NIL, EXPLODE returns NIL (an empty list). If a is a character atom, EXPLODE will return (LIST a). If a is a number or a non-atomic expression, EXPLODE returns an error message and "unwinds."

FIRST (ℓ) SUBR

FIRST finds the first atom or dotted pair on the list ℓ. It uses the function DOTPAIR recursively on the CAR chain of L.

(FUNCTION fn) Special Form (Compiler, Evalquote)

FUNCTION is used to transmit functional arguments. fn can be either the name of a true function (not a Macro or a Special Form) or a LAMBDA or LABEL expression for a function. If fn is a function name, FUNCTION causes a pointer to the compiled code for fn to be transmitted to the calling function (note that in this case, FUNCTION can be omitted and will result in the print-out fn NOT DECLARED but will cause no error in compilation. If FUNCTION is followed by a LAMBDA or LABEL expression, COMP is called to compile the expression under a Gensym name, and a pointer to the resulting compiled code is transmitted to the calling function.

For example:

(LAMBDA (X) (MAPCAR X (FUNCTION ADD1))) ((0 1 2 3)) causes the code pointer for function ADD1 to be transmitted to MAPCAR. On the other hand:

(LAMBDA (X) (MAPCAR X (FUNCTION (LAMBDA (J) (CONS J J)))) ((A B C D)) causes the FUNCTION expression to be compiled and the pointer to the code for the function (LAMBDA (J) (CONS J J) to be transmitted to MAPCAR.

GENSYM () SUBR (Compiler, Evalquote)

Each call to (GENSYM) generates a fresh and distinct atomic symbol of the form GØØØØ1. Gensyms are not placed on the OBLIST and are collected by the garbage collector if they are not in use.

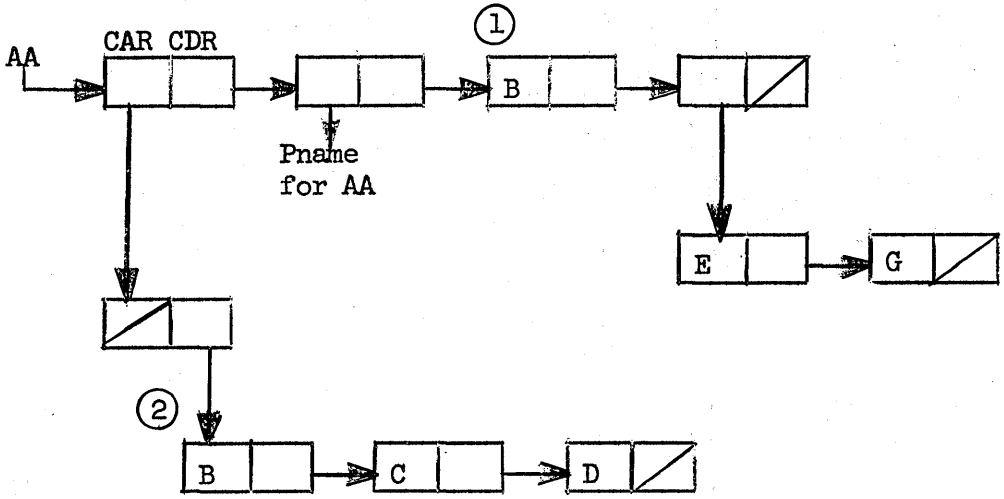
GET (x y) SUBR (Compiler, Evalquote)

GET searches list x for an element EQ to y. If the test succeeds, GET returns the CADR of the list (i.e., the next element on the list). If the test fails, GET returns the value NIL. If x is a quoted atom, then GET searches the property list of x, otherwise it searches the list which is the value of x.

For example, given CSET (AA (B C D))

DEFLIST(((AA(E G))) B)

the structure of the atom AA is



Then (GET (QUOTE AA) (QUOTE B)) searches the list starting at ① and returns (E G) while (GET AA (QUOTE B)) searches the chain starting at ② and returns C.

*GETNO (v) SUBR (Compiler)

*GETNO is a system building (cheating) function which, given a list pointer, returns a pointer to an octal number that is equal to the contents of the cell being pointed to.

LAST (x) SUBR (Compiler)

LAST searches a list x and returns the last element at the top level of the list. It will cause an error if applied to an atom or to a list terminated by a non-NIL atom.

Example:

```
LAST ((A B C))   = C
LAST ((A B (C))) = (C)
LAST (A) = LAST ((C . B)) = LAST ((A B . C)) = error
```

LEFTSHIFT (a b) SUBR (Compiler)

LEFTSHIFT (a b) produces an octal number equal to the integer part of a shifted left by b bits, with zero brought in at the right to replace the shifted bits. If b is negative, a right shift results and zeros are brought in at the left end of the word. If both a and b are negative, the sign of a is not extended, and the resulting value of LEFTSHIFT will be positive. The acceptable range for b is $-47 \le b \le 47$.

*LOCN (a) SUBR (Compiler)

*LOCN (a) produces an octal number equal in value to the pointer a.

(MAP x fn) SUBR functional (Compiler)

MAP applies the function fn to x and to successive CDRs of x until x is reduced to a single atom (usually NIL) which is returned as the value of MAP.

```
(MAP (LAMBDA (X FN) (PROG (M) (SETQ M X)
LP (COND ((ATOM X) (RETURN M))) (FN M) (SETQ M (CDR M)) (GO LP))))
```

MAP cannot be input as the top level function to Evalquote since the functional argument must be evaluated or compiled.

(MAPCAR x fn) SUBR functional (Compiler, Evalquote)

MAPCAR constructs a new list whose value is a list of elements each of which is obtained by applying the function fn to the corresponding element of the list x.

MAPCAR is non-recursive, and uses ATOM to find the end of the list.

MAPCAR cannot be input as the top level function to Evalquote.

Examples of the use of MAPCAR are:

```
(LAMBDA (L) (MAPCAR L (FUNCTION SUB1))) ((Ø 1 2.3)) = (-1 Ø 1.3)
(LAMBDA (L) (MAPCAR L (FUNCTION (LAMBDA (J) (COND ((ATOM J) (QUOTE ATOM))
 (T NIL)))))) ((A B (C) D)) = (ATOM ATOM NIL ATOM)
```

(MAPCON x fn) SUBR functional (Compiler)

MAPCON maps list x onto a new list fn (x) using NCONC, so that the resulting list is formed by concatenation, and uses ATOM to find the end of list x. (MAPCON (LAMBDA (X FN) (COND ((ATOM X) X) (T (NCONC (FN X) (MAPCON (CDR X) FN))))))

MAPCON cannot be input as the top level function to Evalquote because of its functional argument. Also, because of the use of NCONC, MAPCON will damage the system or will cause an endless loop or both, unless the function fn is chosen carefully. (See DELETEL for an example of the use of MAPCON.)

(MAPLIST x fn) SUBR functional (Compiler)

MAPLIST maps the list x onto the list fn (x). It performs the same function as MAP except that it produces an output list by CONSing together all of the results of the form fn (x) computer during the mapping.

MAPLIST is non-recursive, and uses ATOM to find the end of list x. Because of its functional argument, MAPLIST cannot be input as the function at the top level of Evalquote.

Example: (LAMBDA (X) (MAPLIST X (FUNCTION (LAMBDA (J) (CONS (QUOTE B) J))))) ((A B C D)) = ((B A B C D) (B B C D) (B C D) (B D))

MEMBER (a b) SUBR Predicate (Compiler, Evalquote)

MEMBER is a predicate which is true if a is a member of list b, and NIL otherwise. EQUALN used to perform the equality test. Hence MEMBER (1.0 (A B 1 2)) = NIL

NUMBERP (x) SUBR Predicate (Compiler, Evalquote)

NUMBERP (x) is true if x is a pointer to a LISP number, and false otherwise. In particular, NUMBERP (NIL) = NIL.

*PLANT (a b) SUBR (Compiler)

*PLANT is a function used by LAP to plant code in core. It plants the octal quantity corresponding to the value of a into the location corresponding to the value of b.

Thus, *PLANT (ø 4øøø2Q) would change the contents of core location 4øøø2Q to zero (this would wreck the system. *PLANT must be used with caution!)

Anything done by *PLANT could be done by LAP, viz., LAP ((4øøø2Q (ø)) ( )) would achieve the same result. However, the arguments of

*PLANT are subject to normal LISP interpretation, while those of LAP are interpreted in a fashion peculiar to LAP. (Also, LAP can install an entire block of code, while each call to *PLANT changes only one cell).

PRINØ (s) SUBR (Compiler, Evalquote)

PRINØ is used by PRINT to decompose an S-expression S into a string of atoms, parentheses, dots and spaces and calls PRIN1 to fill the print buffer. PRINØ does all of the work of PRINT except for the final (TERPRI). The value of (PRINØ S) is S. To print two S-expressions a and b on the same line, one can use

```
(LAMBDA (A B) (PROG ( ) (PRINØ A) (PRINT B)))) (FIRST SECOND)
```

This will result in the following printout:

```
FIRST SECOND
NIL
```

PRIN1 (a) SUBR (Compiler, Evalquote)

PRIN1 accepts any atom a and packs its print name into the print buffer. It is the only function of the system which packs the output buffer. All other printing functions (such as PRINØ and BLANKS) use PRIN1 as the basic building block. The value of (PRIN1 a) is a.

Example:

PRIN1 (A) results in AA

PRIN1 (ABCD) results in ABCDABCD since the value of PRIN1 is its argument.

```
(LAMBDA (A B) (PROG ( ) (PRIN1 A) (PRIN1 BLANK)
(PRIN1 B) (BLANKS 3) (PRIN1 B) (PRIN1 A) OPRIN1 PERIOD)
(TERPRI))) (FIRST SECOND) results in the following printout:
```

```
FIRST SECOND  SECONDFIRST
NIL  (the value of the PROG)
```

PRINT (s) SUBR (Compiler, Evalquote)

```
(PRINT (LAMBDA (S) (PROG ( ) (PRINØ S) (TERPRI) (RETURN S))))
```

PRINT prints one full S-expression in standard format. (See section 2.2 for the standard format.)

PRINTCH (c) SUBR

If c is a character atom (such as 'A), PRINTCH enters the corresponding character (A) into the print line at the next available byte position. If c is NIL, PRINTCH fires TERPRI. PRINTCH returns c as its value.

(PROP x y fn) SUBR functional (Compiler)

PROP searches the list x for a property EQ to y. If one is found, the value of PROP is a pointer to the CDR of the list. If the property is not found, the value of PROP is (fn), a function of no arguments.

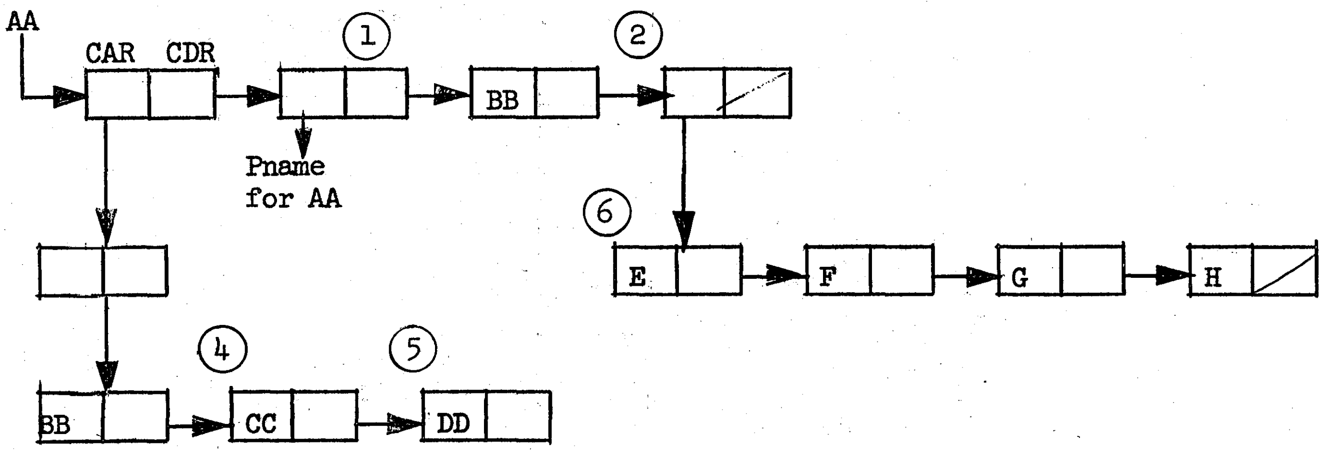
Because of its functional argument, PROP cannot be used as a function at the top level of Evalquote.

For example, given

CSET (AA (BB CC DD))

DEFLIST (((AA (E F G H))) BB)

the structure of the atom AA is



Then
(LAMBDA (X) (PROP (QUOTE AA) Y (FUNCTION (LAMBDA ( ) 2))) (BB)
searches the property list of AA starting at 1 for BB, and returns a pointer to ②, the value ((E F G H)). The same function applied to argument CC cannot find property and hence returns 2 as its value.

On the other hand, the function

(LAMBDA (X) (PROP AA X (FUNCTION (LAMBDA ( ) ( )))) (BB)

searches the value of AA starting at ③ and returns a pointer to ④ or the value (CC DD). The same function applies to value CC returns a pointer to ⑤ with value (DD).

by comparison, since GET when it succeeds returns a pointer to the CADR of what PROP finds, GET (AA BB) yields a pointer to ⑥ or the value (E F G H), while (LAMBDA (X) (GET AA X)) yields the atom CC or DD when applied to BB or CC, respectively.

*RATOM ( ) SUBR (Compiler, Evalquote)

(*RATOM) is the basic LISP reading function, which always returns a single atom whenever called. If the input buffer is empty when (*RATOM) is called or before a delimiter is found, a read command is issued to TSS (2 bells occur on the teletype). Otherwise *RATOM scans the input buffer and returns a single atom, consisting of LPAR, RPAR, PERIOD, a numeric atom, a character atom, or a literal atom. A literal atom not already there is added to the OBLIST.

*RATOM calls ERROR if an illegal character is found outside of a $$ context, or if it finds an illegal format (e.g., a numerical atom in incorrect format).

READ ( ) SUBR (Evalquote)

(READ) calls for one S-expression to be read from teletype, using functions (READ1) and (*RATOM). READ calls ERROR if a right parenthesis or period (not a decimal point) occurs, and calls READ1 every time it sees a left parenthesis.

READ1 ( ) SUBR (Evalquote)

(READ1) is a function used by READ to read a non-atomic S-expression. READ1 is entered after one left parenthesis has been encountered. It calls *RATOM or READ1 successively until the matching right parenthesis is read and calls CONS to tie atoms together appropriately to build the corresponding list structure in core. If an illegal structure is encountered, READ1 produces a diagnostic and calls ERROR.

READCH ( ) SUBR

READCH reads the next character in the input line. The value of READCH is the character atom read. If the read line is empty, READCH fills the buffer from the teletype and then returns the character atom read. READCH does not see the character (77Q) meaning end-of-message, and hence cannot return the character atom 1ØØ77Q as its value.

SET

SET is not implemented.

(SETQ a v) Special Form (Compiler, Evalquote)

SETQ is a special form which evaluated its second argument v and assigns this value, which is also the value of SETQ, to the atom given as its first argument a. In general, a is treated as if it were quoted. If a is not an atom, an error results. If a is not in Special status and is bound in a function by LAMBDA or PROG, SETQ affects only the cell on the pushdown list of the function.

If a is in Special status and has had a previous CSET binding SETQ changes the value of that binding, by being compiled as open code equivalent to (RPLACA (CAR a) v).

If a is in Special status had no previous binding ((CAR A) = NIL) then an error results.

SETQ can be used in series to set many variables to the same value as (SETQ X (SETQ Y Z)) which sets both x and y to the value of z.

SPECIAL (x) SUBR (Compiler, Evalquote)

The argument of SPECIAL, x is a list of literal atoms. SPECIAL sets a flag in bit 4 of the atom head, and returns a copy of its input list.

The Special flag on an atom serves only to tell the compiler that if this atom is bound by LAMBDA or PROG, the old binding of the atom must be saved and the current binding attached to the atom head (rather than to the pushdown list).

SQUOZE (ℓ) SUBR

SQUOZE is a primitive used by COMPRESS below. The value of SQUOZE is a "literal" atom formed from the list of character atoms ℓ. SQUOZE is undefined if ℓ is anything other than a list of character atoms.

SUBST (x y z) SUBR (Compiler, Evalquote)

SUBST substitutes x for each occurrence of the list structure y in the list structure z. The function EQUALN is used to perform the test, so that x, y, and z can have the most general form.

Examples:

```
SUBST (A B (B C E))  =  (A C E)
SUBST (A B (B (B . C) (B)))  =  (A (A . C) (A))
SUBST (A (B) (C B))  =  (C . A)
        since (C B)  =  (C . (B NIL))
```

```
SUBST (A (B) (B C)) =  (B C)
SUBST (2 3 (3 4.5))  =  (2 4.5)
   but
SUBST (2 3 (3.Ø 4.5))  =  (3.Ø 4.5)
   since EQUALN (3 3.Ø)  =  NIL
```

TEREAD ( ) SUBR (Compiler, Evalquote)

(TEREAD) causes the read buffer to be reset so that the next call to READ (or to *RATOM) will ask for new teletype input. If (READ) is called without (TEREAD) and if at the last READ there were any right parentheses left in the buffer, (READ) would call ERROR. (TEREAD) prevents this. The value of TEREAD is NIL.

TERPRI ( ) SUBR (Compiler, Evalquote)

(TERPRI) causes the contents of the print buffer to be printed, induces a line feed, and resets the print buffer. If the print buffer is already empty, (TERPRI) causes a line feed to occur.

Consecutive (TERPRI)s result in skipping print lines. The value of (TERPRI) is NIL.

TRACE (<u>x</u>) SUBR

The argument of TRACE, <u>x</u>, is a list of function names ((f_1 f_2 f_3 ... f_n))

TRACE performs (TRACE (LAMBDA (X) (MAPCAR X (FUNCTION *MKTRC))))

The function *MKTRC tests each of these names f_i in turn. If a function is traceable and is not already in Trace status, *MKTRC changes the instruction (BXH *PDLGN 1 *NDPDL) to a (BUC$_1$ 6 TRACE), and returns the name of the function. If the test fails, *MKTRC returns NIL.

The value of TRACE is therefore a list composed of the names of those functions f_i which were set to Trace status and the remainder of the names replaced by NIL.

If a function (say DIFFERENCE) is used while in Trace status, a typical TRACE printout would be

```
ARGS OF DIFFERENCE
5
3.0

VALUE OF DIFFERENCE
2.0           (any other printing starts here)
```

The function TRACE must be used with caution. It can result in a large amount of printout if used on a recursive function, unless the user intentionally induces a Rescue from TSS to stop printing.

UNSPECIAL (x) SUBR (Evalquote)

UNSPECIAL is similar to SPECIAL. x should be a list of literal atoms. For each atom in x, UNSPECIAL clears bit 4 of the atom head to zero. The value of UNSPECIAL is x.

UNTRACE (x) SUBR

UNTRACE undoes the effect of a previous TRACE.

(UNTRACE (LAMBDA (X) (MAPCAR X (FUNCTION *MKUNT)))) where *MKUNT checks each function named in list x for the instruction (BUC Ø 6 TRACE) and either replaces it with (BXH *PDLGN 1 *NDPDL) and returns the function name, or simply returns NIL.

The value of UNTRACE is a copy of the input list x in which those function names which were not changed to normal status (presumably because they were not being traced) are replaced by NIL.

4.5 ARITHMETIC FUNCTIONS AND PREDICATES

In Q-32 LISP, floating point numbers have the full accuracy available in the 48 bit words: 12 bits of characteric plus 36 bits of mantissa. All approximate tests of equality of floating point numbers use 30 bits of accuracy in the mantissa using function *EQP.

Macros:

The arithmetic Macros are listed in the left hand column below:

MAX	*MAX (x y)
MIN	*MIN (x y)
LOGOR	*LOGOR (x y)
LOGAND	*LOGAND (x y)
LOGXOR	*LOGXOR (x y)
PLUS	*PLUS (x y)
TIMES	*TIMES (x y)

Each of the above Macros is defined in terms of the corresponding simple function whose name begins with an asterisk. The simple arithmetic functions have exactly two arguments. For example, the definition of MAX is

```
MACRO (((MAX (LAMBDA (L) (*EXPAND L (QUOTE *MAX)))))).
```

The arithmetic of PLUS and TIMES is floating point if any arguments are floating, integer otherwise. PLUS never produces -Ø as an output. LOGOR, LOGAND and LOGXOR fix their arguments and produce octal integers as answers. The other functions all produce answers in floating point or decimal integer format.

Other Arithmetic Functions

ABSVAL (x)

Compiles the absolute value of number x.

ADD1 (x)

Adds 1 to X. (ADD1 (LAMBDA (x) (PLUS x 1)))

DIFFERENCE (x y) also DIFFER (x y)

Subtracts y from x. (DIFFERENCE (LAMBDA (X Y) (PLUS X (MINUS Y))))

DIVIDE (x y)

Divide x by y uses subroutine *DIVIDE. Division is in floating point if either x or y is floating; integer otherwise. DIVIDE forms a list of QUOTIENT and REMAINDER.

ENTIER (x)

Computes the integer part of x for positive x and - integer part of -x for negative x.

EXP (x)

Computes e^x, using functions EXPT and *EXPTF.

*EXPF (x)

Computes e^x for $-1 \leq x \leq 1$. Used by EXP.

EXPT (x y)

Raises x to the y power. The result is an integer if x is an integer and y is a positive integer and if the value is less than 2^{47}; otherwise the value is a floating point number. Functions LOG, EXP, and *EXPTI are used for some cases.

*EXPTI(x y)

Raises x to the y power by power product. y must be a positive integer. Computation is done in floating point or integer arithmetic, depending upon the representation of x.

FLOAT (x)

Produces a floating point output equal to the input x.

JUST (x)

Reduces a number x modulo 2^{18}.

(JUST (LAMBDA (X) (*LOGAND 777777Q)))

LOG (x)

Computes the natural logarithm of x, for positive x, and gives an error diagnostic for x ≤ Ø.

MINUS (x)

MINUS produces -x as its value.

QUOTIENT (x y)

For fixed-point arguments, the value is the number theoretic quotient. If either X or Y is a floating point number, the answer is the floating point quotient.

REMAINDER (x y)

Computes the number theoretic remainder for fixed-point arguments, and floating point residue for floating-point arguments.

SQRT (x)

SQRT takes a floating point square root of the absolute value of x. with no check as to original sign of x.

SUB1 (x)

Subtracts 1 from X.

(SUB1 (LAMBDA (x) (PLUS x -1)))

Arithmetic Predicates:

*EQN (x y)

Tests two numbers for equality of representation. Thus

```
*EQN (Ø  -Ø)  =  NIL                 *EQN (Ø    Ø) = T
*EQN (1    1Q)  =  NIL               *EQN (-Ø  -Ø) = T
*EQN (1   1.Ø)  =  NIL               *EQN (1.Ø 1.Ø) = T
*EQN (1Q  1.Ø)  =  NIL               *EQN (1Q   1Q) = T
*EQN (3Q4 3ØQ3) =  T                 *EQN (1.Ø 1.ØEØ) = T
*EQN (1.Ø  1.ØØØØØØØØØØ1) = T
```

The last case holds because the last decimal place is lost in the internal representation.

*EQP (x y)

Tests two numbers for approximate equality. If the numbers are integers, the test is for equality of value (Ø = -Ø). If either of the two numbers is a floating point number, the test is made on equality of all but the right-most 6 bits.

In general:

$$\text{*EQP (x y)} = \begin{cases} \text{T if } \left| \dfrac{x - y}{x + y} \right| < Ø.7 \times 1Ø^{-9} \\ \text{NIL otherwise} \end{cases}$$

FIXP (x)

Is true if x is a fixed point number, an error if x is not a number, and false if x is floating.

FLOATP (x)

Is true if x is a floating point number, an error if x is not a number, and false otherwise.

GREATERP (x y)

True if x > y and false if x ≤ y.

Note that it is possible for both *EQP (x y) and GREATERP (x y) to be true simultaneously, but (AND (LESSP X Y) (GREATERP X Y)) is always NIL.

LESSP (x y)

This is true if x < y and false if x ≥ y.

Note that it is possible for both *EQP (x y) and LESSP (x y) to be true if either x or y is floating.

MINUSP (x)

Tests whether x is negative.

ZEROP (x)

(ZEROP (LAMBDA (X) (*EQP X Ø)))

ZEROP (x) is true if X = Ø or X = -Ø and NIL otherwise.

4.6 BUFFER FUNCTIONS AND SAVE

LISP arrays are not implemented in Q-32 LISP. There exists two functions for the handling of buffers or arrays of non-pointer data, as follows:

GETBUF (m n)

GETBUF creates a non-pointer array or buffer of the specified number of words n. The name m of the array is any legal atom and is the value of the procedure. The number of words n is limited only by the availability of free and full word space at the time of procedure call.

SETBUF (m n)

SETBUF sets every word of the array named m to the specified constant n. Useful for clearing or initializing an array. The array name m is the value of the procedure.

SAVE (n)

SAVE is a LISP function that saves current core contents (essentially an array core dump) on magnetic tape reel n. This dump is in a form compatible with the Time-Sharing System's LOAD command.

SAVE makes its own binary tape request from the Time-Sharing System, so that GETFILE is not used before SAVE. However, DEFILE should be used following a SAVE to release the tape drive. To save the LISP system on a scratch tape, the user should type SAVE (Ø). SAVE, like GETFILE, will cause TSS to reply $WAIT, and later $FILE n ØØ DRIVE d REEL n. After SAVE is completed, LISP will reply n, then 2 bells. If a scratch tape was used for the SAVE, the user will have to dial the operator to save that reel and give him the name of the user and title of the tape.

CAUTION:

Saving a LISP system is at the user's risk. A saved system will not be updated along with standard LISP, and may have to be loaded from tape rather than disk. In general, it is desirable to save a library tape of S-expressions to be loaded into LISP, in addition to saving the version that is in core.

References

1. The Programming Language LISP : Its Operation and Applications. Information International, Incorporated, Cambridge, Mass. March 1964

2. LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass. August 1964

3. Clark Weissman. LISP Primer: A Self-Tutor for Q-32 LISP 1.5. SDC Series TM-2337/010/00. June 1965

4. Command Research Laboratory User's Guide. SDC Series TM-1354

5. S. L. Kameny. LISP 1.5 Input-Output File and Library Functions. SDC Series TM-2337/102/00. September 1965.