YKTLISP Program Description and Operations Manual

January 5th, 1983

Various, highly modified by Cyril W. Alberg

Computer Science Department
IBM Thomas J. Watson Research Center
P. O. Box 218, Yorktown Heights, New York 10598
This is a manual in process. As of this date it is incomplete, but slowly growing. As each new edition is produced, this section will be updated with an outline of the changes from earlier versions.

The section "Table of system functions, variables and commands" on page 145 contains very telegraphic descriptions of all the operators in the system which are not meant purely for system use. Those which have more complete descriptions in the body of the manual will have one or more page references to those descriptions.

The base manual, to which all changes will be related, will be that of November 22, 1982.

December 7, 1982.

Added short descriptions of operators only available in the LISPnnnn systems.
Added chapter on debugging.

January 5, 1983.

General clean up of misspellings, format errors, inconsistent notation.
Added chapters on LISPLIBs and compiler environment.
This manual attempts to describe the YKTLISP system, with the exception of LISPEDIT. It has been produced by merging and rewriting the LISP/370 Program Description/Operations Manual (SH20-2076-0), the Research Report LISP/370 Concepts and Facilities, RC 7771, by Fred W. Blair, and various internal notes and memos.

LISPEDIT is documented by a set of on-line command descriptions, which are also available, with added commentary, from Martian Mikelsons, IBM Research, Yorktown Heights.
CONTENTS

Introduction .................................................. 1
How to access YKTLISP from CMS .......................... 3
Loading and running YKTLISP .............................. 5
Loader internals .................................................. 7
Invoking the loader ............................................. 9
Invoking the unloader ......................................... 13
Using a saved system ......................................... 15
Interaction with YKTLISP ...................................... 17
YKTLISP programs .............................................. 19

YKTLISP syntax and semantics ............................... 23
YKTLISP data types ............................................ 25
Identifiers ....................................................... 24
Pairs ........................................................................ 25
Lists ....................................................................... 26
Numbers .................................................................. 28
Vectors .................................................................. 29
  Reference, Word and Real Vectors ....................... 30
  Character Vectors .............................................. 30
  Bit Vectors ....................................................... 31
Binary program images ......................................... 32
Funargs .................................................................. 32
State descriptor .................................................... 33
Streams ............................................................... 33
  Character streams .............................................. 33
  Key addressed streams ...................................... 35

Expressions: functions, macros and special forms ....... 37
Expressions ........................................................ 37
Constants ............................................................ 37
Variables ............................................................. 37
Lists as expressions ............................................. 37

Evaluation ............................................................ 39
The environment of evaluation and execution ............ 40
  Stack frames ...................................................... 41
  Bindings and variable referencing ....................... 41
  The environment of compilation ......................... 42

Descriptions of the operators of YKTLISP ............... 43

A note on naming conventions ................................ 45

Environment of execution ..................................... 47
Specification of values ........................................ 47
Sequence of evaluation ........................................ 48
Conditional evaluation ........................................ 50
Function and macro definition, variable binding ....... 52
Multiple level returns ......................................... 55

Environment of evaluation .................................... 59
Evaluation .......................................................... 59
Assignment ........................................................ 61

Iteration over lists and vectors ............................ 63
MACLISP style operators ....................................... 63
MACLISP style operators for vectors ..................... 66
LISP 1.5 style mapping operators ......................... 68
Miscellaneous ....................................................... 68
Auxiliary operators .............................................. 69

Data types, type testing and other predicates ......... 71
General .............................................................. 71

Contents vii
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Relation between SHLISPWS and memory</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Error return codes from the loader</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Error return codes from YKLISP ABENDs</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Box representation of pairs</td>
<td>25</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Box representation of list, corresponding to dot notation</td>
<td>26</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Box representation of list, corresponding to list notation</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Box representation of a cyclic list</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Box representation of equivalent shared and non-shared lists</td>
<td>28</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Box representation of a reference vector</td>
<td>30</td>
</tr>
<tr>
<td>Figure 10</td>
<td>A list, interpreted as a character stream</td>
<td>34</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Effect of NEXT and WRITE on stream</td>
<td>34</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Description of a fictitious operator</td>
<td>43</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Abbreviations for operand data types</td>
<td>44</td>
</tr>
<tr>
<td>Figure 14</td>
<td>CATCH, MESS values, actions and interpretation</td>
<td>56</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Result of MAPLIST</td>
<td>65</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Result of MMAPCIN</td>
<td>65</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Result of LOTSOF</td>
<td>81</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Result of APPEND</td>
<td>82</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Effect of NCONC</td>
<td>89</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Effect of NREVERSE</td>
<td>89</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Character string allocation</td>
<td>95</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Bit string allocation</td>
<td>96</td>
</tr>
</tbody>
</table>
INTRODUCTION

This manual is intended as a guide to the facilities and capabilities of YKTILISP. It contains reference material describing the functions available in that system, as well as material comprising a system programmer's guide. It also contains a certain amount of tutorial material that provides some motivation or explanation for why certain operations are performed in the way they are.

This manual is not intended as a basic primer for LISP. For that purpose, the reader should consult another publication such as Let's Talk LISP by Laurent Sirkossy (Prentice-Hall, 1976), The Programmer's Introduction to LISP by W. D. Maurer (Elsevier, 1972), or LISP by F. H. Winston and B. K. P. Horn (Addison Wesley, 1981), all of which are textbooks presenting an introduction to LISP for the beginning LISP programmer. Other books, such as Artificial Intelligence by Patrick Winston (Addison Wesley, 1977) and Computational Semantics by E. Charniak and Y. Wilks (Elsevier, 1976) contain chapters introducing LISP in the course of examining some of the application areas where LISP programs have been significant.

This LISP system was originally developed in the VM/CMS programming environment, and this has affected the structure and facilities included in the implementation. Nevertheless, we have tried to avoid any real dependencies on features unique to that environment. The original LISP/370 was provided with a MVS/TSO system interface. This has not been maintained in YKTILISP, however the system dependent code is even more loosely coupled to LISP than in LISP/370, making the task of providing such an interface no harder, and possibly easier than in the earlier system.

The version of YKTILISP documented here is one of a series of systems produced during the continuing development of LISP at the IBM Thomas J. Watson Research Center. This version was selected for submission as an Installed User Program because it had been used for more than a year at that site, during which time we felt that most major implementation errors had been detected and corrected. In fact, a sizable number of errors remained in the system as released. In addition, since the time when this system was devised, our thoughts about several aspects of LISP have evolved. The current system embodies both corrections for many of the errors extant in the IUP, but also carries the system several steps along the road to our current ideal.

Use of a screen display console, such as the IBM 3270 series of devices, is recommended for program development in the YKTILISP system. Normal test and debugging activity profits greatly from the rapid display capability of these devices. Once a LISP application has been developed, the value of this type of terminal will depend upon the application itself rather than any characteristic of YKTILISP.

YKTILISP contains a powerful, screen oriented, structural editor, LISPEDIT, which will aid the user in the development of his systems. LISPEDIT is separately documented.

There still remain a number of extensions which should be added to the system. These include: a faster, non-redefinable function linkage (suspiciously like other LISP's function value cell); a non-garbage collected area of heap; redefinition of the FR and MR objects as non-identifiers; a CASE primitive; etc. It is not clear when, or even if, any of these additions may be made.
YKTLISP cannot be loaded by either of the CMS modules LOAD or LOADMOD, as storage must be dynamically allocated, and data pointers relocated in ways peculiar to LISP. Thus, transient routines (modules) exist to load and unload YKTLISP. A third module is required, LISP CMS, which contains all of the VM dependant code for YKTLISP. Finally, the code and data which constitute YKTLISP itself are stored in two files, of filetypes SEGMENT and SHLISPWS.

We use stereotyped names for these files, eight character with the last four numeric, like GOLD0026, BASE0037, LEDT0052, etc. In these names the numeric portion is a version number, completely non-automatically maintained, which is used to coordinate generations of the system.

The five distinct files which make up a YKTLISP system are:

aaaaaaa SHLISPWS The code and data which cannot be in read-only storage, belonging to both the system and the user

bbbbbbb SEGMENT The read-only portion of the system. On a VM system which has a discontinuous shared segment for LISP this file may not be needed in particular cases.

LISPCMS MODULE The VM system dependant code acts as an interface between the LISP world and the operating system.

WARMnnnn MODULE The loader for the three preceding files.

DROPnnnn MODULE The "unloader", releases storage which was allocated during loading. (Not really needed, you could always re-IPL CMS or HX, or LOGOFF, for that matter.)

As an example, with no EXEC, we will assume the following files are on accessed disks:

WARM0045 MODULE
DROP0045 MODULE
LISPCMS MODULE
LISP0052 SEGMENT
LEDT0052 SHLISPWS

Then, to use YKTLISP, the minimum in typing would be:

WARM0045 LEDT0052 LISP

At this point you should be in the YKTLISP read-eval-print loop.

At the Yorktown research center all the files need to load and run YKTLISP are located on two mini-disks, ALBERGA 197 and MIKELSN 196. In addition there is an EXEC, YKTLISP, on the CMSSYS 19F disk which will make the needed links and invoke the loader. Thus, on any of those systems it is sufficient to type:

LISPEDIT

to use the system.

For further information on the YKTLISP EXEC, enter YKTLISP ? on any of the Yorktown VM systems.

When ever you are communicating with a system read-eval-print loop, (there are two, the supervisor, SUPV, and the break-loop), entering (FIN) will signal your desire to end the loop. If you are at the "top-level" this will return control to CMS. Alternatively, execution of the LISP function (RET) will produce an immediate return to the caller of LISP. (RET) may be evaluated at any level, it need not be the top level supervisor, to leave LISP. (Of course you can always attention yourself to VM and HX, or to CP and re-IPL, but that doesn't count.) YKTLISP will still be loaded in memory, holding what ever storage you might have allowed it. The com-
mand LISP will re-start it, with all your work still present. If you wish to use your entire virtual machine for something else you must unload YKTLISP. That is accomplished simply by entering the command:

```
DROP0045
```

If, during the course of a session, you have defined or compiled some functions, or have built data structures in memory, and you wish to preserve this work you may invoke the function FILELISP, which will create a new SHLISPWS file on disk. This new file can be loaded as "BASE0052" was above, and you will return to the state which obtained at the time of FILELISPing.

This "script" is the simplest case (for the given file names).

In real life matters become rather more complex. To make what follows intelligible I must digress into certain aspects of YKTLISP internal organization.
The code and data which comprise a loaded YKTLISP system reside in three potentially contiguous areas of memory, a Lisp-user area, a Lisp-system area, and a system-dependent area, corresponding to the SHLISPWS, SEGMENT, and LISPFCMS MODULE files respectively. The Lisp-user area is in turn sub-divided into six regions, NILSEC, Heap1, Heap2, Stack1, Stack2 and UBPI. The Lisp-system area has two regions, FIXEDSEC and SBPI.

NILSEC contains various data which are not moved by the garbage collector, and acts as a communication area between separately assembled and/or loaded functions.

HEAP & STACK contain the Lisp data and the "frames" for the Lisp interpreter and compiled functions. The duplicating of these areas is necessitated by the garbage collection algorithm chosen for YKTLISP.

UBPI (User Binary Program Image) contains compiled and assembled functions linked or loaded after the creation of the corresponding SEGMENT file.

FIXEDSEC contains the portions of the system written in assembler-H. These include the garbage collector, the interpreter, the function linkage code, as well as a large collection of built-in functions.

SBPI contains the portions of the system written in Lisp and/or LISP.

The SHLISPWS files contain copies of the contents of NILSEC, UBPI, and the active HEAP and STACK for a YKTLISP system, as they existed at the time of the call to FILELISP which created the file. They also contain a header record (which is obtained from the literal data for the Lisp EXEC) which gives the minimum storage (in bytes) needed for each such area. Each SHLISPWS file is associated with a specific SEGMENT file.

The system is designed to use a contiguous shared segment (DSS) for the preferred SEGMENT. At Yorktown, such a DSS exists, and is named LISP. The loader and unloader modules have this DSS name built in. At any other installation there may be no shared segment, or if there is one it may have another name. In the latter case the loader and unloader must be re-assembled, with the DSS name replaced. In either case, keep that in mind when reading the following description.

In referring to SHLISPWS and SEGMENTS, the word "file" will be dropped at this point, with SHLISPWS always referring to a file, and SEGMENT to either a file or a contiguous shared segment, which ever pertains.

Since all compiled functions, including those in SEGMENTS, use NILSEC as a communication area, it is impossible to load a SHLISPWS with a SEGMENT other than the one which was loaded at the time the SHLISPWS was created, and to expect the system to operate. In order to preclude such actions each SEGMENT contains a sixteen byte key, consisting of the machine-id of the computer on which it was created, concatenated with the time-of-day clock value at its creation time. Each SHLISPWS contains, in its header the key from the SEGMENT which was loaded when it was created, as well as the file-name under which that SEGMENT was stored.

The loader first uses DMSPREE to reserve part of the virtual memory for non-LISP uses, such as disk directories, editors, etc. It then locates and loads, into DMSPREE high storage, the LISPFCMS MODULE. If the load request (see options, below) is for system storage the virtual memory size is found and the existence of a contiguous shared segment (named LISP) is tested for. If the USERSTOR/SYSSTOR parameter (see WARMnparameter description, below) requests SYSSTOR, and if a DSS exists, its starting location is compared with the memory size to see if it can be used, and if it can it is loaded, either in shared or nonshared mode, depending on the parameters. If no DSS exists, or if it overlaps the user's virtual memory, the SEGMENT will be loaded into user storage, and the processing skips the
next section. The storage keys are then set in the shared segment area, defaulting to D, which will prevent modification. The header record from the SHLISPWS if now read.

If the DSS has been loaded, its sixteen byte key is compared with the key in the header record of the SHLISPWS. If they do not match the length of the proper SEGMENT (saved in the SHLISPWS header) is compared against the length of the DSS. If the proper SEGMENT is not longer than the DSS, the DSS is re-loaded in non-shared mode. Otherwise the DSS is purged and the USERSTOR/SYSSTOR flag is forced to USERSTOR. The minimum amount of space required by the SHLISPWS is computed including any absolute size parameters (and including the size of the corresponding SEGMENT, if USERSTOR is requested, if no DSS exists, or if the DSS area is unusable for any reason) and a DMSFREE is issued with that as its minimum and 16M as its maximum. Once we have all the rest of the machine we release the DMSFREE space we grabbed at the very beginning.

We now divide up the allocated space among the various areas and regions. Each region is given an absolute amount of space, the sum of its current size (from the header) and the size parameter. Any remaining space is then partitioned on the basis of the percentage requests.

Finally, the files are loaded, first the SEGMENT if the DSS did not exist, did not match the SHLISPWS, or if USERSTOR was requested. In these cases the file name is extracted from the SHLISPWS header. Then the SHLISPWS itself is loaded into the user's area, and a scan is made through memory, relocating all pointers to their new locations. An attempt is made to establish a nucleus extension for the loaded system. If the YKTSYS package (or VM/SP) is not present a module file is written (on the user's A disk) which will transfer control to the entry point.
INVOKING THE LOADER

The format for the loader call is (with [...] enclosing optional fields)

```
WARMnnnn [fn [ft [fm]]]
   [[[[NONSHARE | SHARE] [[USERSTOR | SYSTOR]]
      [NIL [nilsz] [nil%]] [BPI [bpisz] [bpi%]]
      [STACK [stkwsz] [stk%]] [HEAP [heapsz] [heap%]]
      [FIXED [fixedsz] [fixed%]] [GETMIN getmzsiz]
      [CMSHIGH cmshsz] [COMMAND command] [KEY key] ()]]
```

where the default values are:

```
WARMnnnn LISP SHLISPWS K (NIL 16K BPI 32K STACK 8K 10% HEAP 8K 90%
GETMIN 320K CMSHIGH 144K COMMAND LISP
KEY 14 SHARE SYSTOR)
```

Note that all but the fn, ft and fm parameters are (or follow) key words, that the order is irrelevant (see the above example, where SHARE and SYSTOR are at the end in the default listing) and that ALL parameters are optional.

- **SHARE**: DSS is loaded in shared mode if it matches the SHLISPWS and USERSTOR option is not present. This is the default.
- **NONSHARE**: DSS is loaded in non-shared mode.
- **SYSTOR**: SEGMENT is loaded into user’s virtual memory. (Note that SHARE forces SYSTOR, while USERSTOR forces NONSHARE. The parameters are scanned from left to right, the last prevailing.)
- **KEY**: The following parameter must be a number, which becomes the storage protection key for the DSS. (Note that CP will allow you to set the storage key in a DSS without affecting sharing.)
- **NIL**: Followed by an absolute and/or a percentage space request for allocation beyond the current contents of NILSEC in the SHLISPWS. Absolute requests are of the form "number", "number"K or "number"M (e.g. 1576K or 3M), where K and M indicate multipliers of 1024 and 1048576 respectively. Percentage requests are of the form "number"%, where "number" should be less than or equal to 100.
- **BPI**: Space requests (a la NIL) for user’s compiled function area.
- **STACK**: Space requests for stack frame area.
- **HEAP**: Space requests for data area. Note that this and STACK are effectively doubled, that is there will be two equal areas allocated, and this is the request for one of them.
- **FIXED**: Space requests for the SEGMENT area. This parameter is ignored if an DSS is available and used. If USERSTOR is specified, or if no DSS exists, it applies as above.
- **GETMIN**: Space request (formatted like the absolute requests, above) for the area to be set aside before the SHLISPWS is loaded.
- **CMSHIGH**: Space request for the area of DMSFREE high memory to be set aside before loading.
- **COMMAND**: Followed by the name to be used as the entry to YKTLISP. This will either become a nucleus extension (see NUCK memo) of the filename of the MODULE written on the user’s A disk.
Figure 1. Relation between SHLISPWS and memory

Figure 1 may help. The left side represents the SHLISPWS file and the SEGMENT file, while the right side represents the computer storage layout.

If you are using a relatively small virtual memory the default values for GETMIN and CMSGHIGH will probably be too large. There is no simple rule to determine "good" values for these parameters. A period of experimentation may be needed to settle on numbers satisfactory in a given configuration. This becomes particularly true when no shared-segment is defined, and the FIXEDSEC and SBPI must be loaded into the user's own virtual storage.
<table>
<thead>
<tr>
<th>RC</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not enough free storage for CMSSHGH.</td>
</tr>
<tr>
<td>2</td>
<td>Not enough getmain storage for MINGET.</td>
</tr>
<tr>
<td>3</td>
<td>LISP CMS MODULE not found.</td>
</tr>
<tr>
<td>4</td>
<td>Not enough free storage for LISP CMS MODULE.</td>
</tr>
<tr>
<td>5</td>
<td>Read of LISP CMS MODULE failed.</td>
</tr>
<tr>
<td>6</td>
<td>LOADSYS of shared segment failed.</td>
</tr>
<tr>
<td>7</td>
<td>Coldstart TEXT deck not found.</td>
</tr>
<tr>
<td>8</td>
<td>More than 100% of available space requested.</td>
</tr>
<tr>
<td>9</td>
<td>SEGMENT file not found.</td>
</tr>
<tr>
<td>10</td>
<td>SEGMENT bigger than shared segment.</td>
</tr>
<tr>
<td>11</td>
<td>Failure writing LISP MODULE.</td>
</tr>
<tr>
<td>12</td>
<td>SHLISPWS file not found.</td>
</tr>
<tr>
<td>13</td>
<td>Read error on work space file.</td>
</tr>
<tr>
<td>14</td>
<td>Loader dump pointer table full.</td>
</tr>
<tr>
<td>15</td>
<td>ADCON which points out of LISP found.</td>
</tr>
<tr>
<td>16</td>
<td>A mixed vector is embedded in another.</td>
</tr>
<tr>
<td>17</td>
<td>Marked non-stored object.</td>
</tr>
<tr>
<td>18</td>
<td>Unknown vector length code type (x2F).</td>
</tr>
<tr>
<td>19</td>
<td>Illegal type found.</td>
</tr>
<tr>
<td>20</td>
<td>Bad vector length encountered.</td>
</tr>
<tr>
<td>21</td>
<td>Error in frontier computation.</td>
</tr>
<tr>
<td>22</td>
<td>NIL full.</td>
</tr>
<tr>
<td>23</td>
<td>HEAP full.</td>
</tr>
<tr>
<td>24</td>
<td>STACK full.</td>
</tr>
<tr>
<td>25</td>
<td>Error in CMS loader.</td>
</tr>
<tr>
<td>26</td>
<td>Insufficient storage for YKLISP.</td>
</tr>
<tr>
<td>27</td>
<td>Error reading descriptor record of SHLISPWS.</td>
</tr>
<tr>
<td>28</td>
<td>Invalid WARMmmm command format.</td>
</tr>
<tr>
<td>29</td>
<td>Error reading shared segment image.</td>
</tr>
<tr>
<td>30</td>
<td>SEGMENT file named in SHLISPWS has wrong key.</td>
</tr>
</tbody>
</table>

Figure 2. Error return codes from the loader.

Various conditions can cause the loading operation to fail. In these cases a value in the range 1 to 30 will be returned to CMS.

<table>
<thead>
<tr>
<th>RC</th>
<th>Process</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>8001</td>
<td>START-UP</td>
<td>Start code unknown.</td>
</tr>
<tr>
<td>8002</td>
<td>COLDSTART</td>
<td>Bad initial global a-list</td>
</tr>
<tr>
<td>8003</td>
<td>COLDSTART</td>
<td>Bad identifier</td>
</tr>
<tr>
<td>8004</td>
<td>COLDSTART</td>
<td>Bad identifier</td>
</tr>
<tr>
<td>8005</td>
<td>COLDSTART</td>
<td>Qbase array too small</td>
</tr>
<tr>
<td>8006</td>
<td>START-UP</td>
<td>No restart state.</td>
</tr>
<tr>
<td>8007</td>
<td>START-UP</td>
<td>Interpreter returned from a SD apply</td>
</tr>
<tr>
<td>8008</td>
<td>START-UP</td>
<td>The SEGMENT image doesn't match the WS.</td>
</tr>
<tr>
<td>8010</td>
<td>RECLAIM</td>
<td>Stack exhausted</td>
</tr>
<tr>
<td>8011</td>
<td>RECLAIM</td>
<td>Keep exhausted</td>
</tr>
<tr>
<td>8012</td>
<td>RECLAIM</td>
<td>Heap exhausted (no room for GC work area)</td>
</tr>
<tr>
<td>8013</td>
<td>EXIT</td>
<td>&quot;Impossible&quot; case</td>
</tr>
<tr>
<td>8014</td>
<td>FINDBIND</td>
<td>Bad display</td>
</tr>
<tr>
<td>8015</td>
<td>BINDER</td>
<td>NILSEC exhausted</td>
</tr>
<tr>
<td>8016</td>
<td>UNBIND</td>
<td>Shallow cell not found.</td>
</tr>
<tr>
<td>8017</td>
<td>GLOBAL</td>
<td>Ill formed global a-list</td>
</tr>
<tr>
<td>8020</td>
<td>SECD</td>
<td>MR CASE used, SECD rules not yet defined.</td>
</tr>
<tr>
<td>8021</td>
<td>SECD</td>
<td>Fatal error in JAMT</td>
</tr>
<tr>
<td>8030</td>
<td>RECLAIM</td>
<td>Qbase array full</td>
</tr>
<tr>
<td>8031</td>
<td>RECLAIM</td>
<td>Heap overflow</td>
</tr>
<tr>
<td>8032</td>
<td>RECLAIM</td>
<td>Unknown type, in vector row</td>
</tr>
<tr>
<td>8033</td>
<td>RECLAIM</td>
<td>Unknown type, 98-9F range</td>
</tr>
<tr>
<td>8034</td>
<td>RECLAIM</td>
<td>Length in vector bad</td>
</tr>
<tr>
<td>8035</td>
<td>RECLAIM</td>
<td>Tangled stack -- ERR10</td>
</tr>
<tr>
<td>8036</td>
<td>RECLAIM</td>
<td>Tangled stack -- ERR9</td>
</tr>
<tr>
<td>8037</td>
<td>RECLAIM</td>
<td>Tangled stack -- ERR7</td>
</tr>
<tr>
<td>8038</td>
<td>RECLAIM</td>
<td>Tangled stack -- ERR8</td>
</tr>
<tr>
<td>8040</td>
<td>GENSYM</td>
<td>Eco-death</td>
</tr>
<tr>
<td>8041</td>
<td>THROW</td>
<td>Bad stack frame encountered.</td>
</tr>
</tbody>
</table>

Figure 3. Error return codes from YKLISP ABENDS.
There exist a number of fatal errors which can occur while YKTLISP is running. These will result in a return to CMS with a return code in the 80xx range.
The format for the unloader call is:

```
DROPnnnn <command>
```

where command defaults to LISP.

The unloader looks for a nucleus extension with the given command name. If it finds one (which presupposes that NUCX is present) it purges it from the nucleus extension table and frees the storage which the loader grabbed (using data stuffed into the first page of NILSEC by the loader). If no nucleus extension is found the A disk is searched for a MODULE file with the command name, which is read. The unloader knows the "shape" of the MODULE, and extracts from it the address of the NILSEC, together with the sixteen byte key inserted there by the loader. If the sixteen bytes in NILSEC which should contain the key match that from the MODULE the storage is released, as above. The reason for this extra check is that the MODULE file can survive a HX or re-IPL, which would release the in-core part of the system.
If the user has saved a YKLISP system file image by using the FILELISP function, he may load that saved system by specifying the file identifier as an argument of the WARMnnnn module. For example, if the YKLISP system file image is named ASK SHLISPWS, it may be invoked by the command:

```
WARM0045 ASK LISP
```

When such a saved YKLISP system is invoked, execution will continue with a return from the function FILELISP which saved the system. This may or may not be the top-level supervisor which receives control when the default system LISPnnnn SHLISPWS is invoked, depending upon the manner in which that particular YKLISP system was saved.
INTERACTION WITH YKTLISP

When you first start YKTLISP (not in LISPEDIT), you are confronted with an EVAL supervisor. This is a program, SUPV, which reads an expression from the console, displays the expression back to you, evaluates it, using the interpreter, and displays its value. It then goes back to the READ, to wait for a new input.

The first thing it will do is print

\[ \text{Value} = n \]

for some value of \( n \). This is simply the number of generations of FILELISP that this particular system has been through.

The echoing of input and the display of values are turned on or off, independently, by the operators SET-ECHO-PRINT and SET-VALUE-PRINT. Called with arguments of () they turn off the appropriate operator, called with non-() arguments they turn it on.

The operators LAST-VALUE and LAST-EXP will return the last value produced, or the last expression read. Thus if you entered an expression and want to save the value you can type

\[ (\text{SETQ } X \ (\text{LAST-VALUE})). \]

If an error is signaled by any process, or if you force an external interrupt, you will be put in the "break loop". This is similar to SUPV. It differs chiefly in not echoing input, and not prefacing the values with

\[ \text{Value} = \]

LAST-EXP and LAST-VALUE do not work in the break loop. Furthermore, if you enter a single ? the break loop will repeat the original error message.

In either SUPV or the break loop entering a "null" line (hitting the ENTER key without any input) will result in a message indicating where you are. SUPV prints "LISP", the break loop prints "BREAK". Immediately upon being started, the break loop clears any stacked lines in the console input and holds them in a variable STACK. This can be examined using &I, but there is no provision for re-stacking them.

To exit from the break loop you must either enter (FIN EXP) or (UNWIND s-int). Many entries to the break loop are not re-startable, and FINing in those cases is equivalent to (UNWIND 1). If an error is recoverable, the error message may tell you what you should do. (Not always though; there are still a lot of loose ends around.) If you FIN the value (of exp) will be tested against a filter set by the operator which reported the error. If it passes, all well and good. If not you will receive a message, such as, "... FIN with small integer, ... Repeat with correct type." "Still in break loop."

One characteristic of READ operator should be pointed out here. Since READ treats its input as a continuous stream of characters, ignoring end-of-lines, any input to either SUPV or the break loop must end with a delimiter. The closing parenthesis of a list, or > of a vector will do, but if you wish to enter a simple identifier or a number you must explicitly type a trailing blank before you hit ENTER, to inform the system that there are no more characters to follow.
Ideally, the descriptions and definitions of the components of a programming language should be written without forward references. No feature should be mentioned before it is defined. While such descriptions may be possible in completely formal definitions, (and not even there when recursive definitions are needed), they are usually close to incomprehensible. The reader requires examples, and the examples almost always require some features other than the one under immediate examination.

Thus we will start by waving our hands a bit, giving a very informal description of certain parts of the LISP language as embodied in YKTLISP. This will provide a framework on which the more exact definitions which follow can be placed. Further, the current discussion will provide a modicum of rational for some of the structure of the language.

In all that follows the reader should try to remember that, unlike most languages, the sequences of character which we use to represent expressions are not the expressions. The expressions are the internal structures of LISP data objects, pairs, identifiers, etc., which are constructed by the LISP reader from these sequences of characters. The interpreter (and the compiler) never see what we see. They act on internal data structures only. It is often useful to describe the data structures of the expression by drawing box diagrams of the structures, although this is never strictly necessary. The internal forms exactly mirror the representations, and one quickly learns to "see" the real structure when looking at a collection of parentheses. (It has been claimed that LISP stands for Long Incomprehensible Sequences of Parentheses.) It is this fact that gives LISP much of its power. Since LISP programs are well formed LISP data they can be easily constructed by other programs.

As was stated in "Expressions" on page 37 a list, (THIS IS A LIST), is interpreted as an expression, with and operator, THIS, and operands, IS, A and LIST.

In LISP every expression has a value. In particular the LISP equivalent of an IF ... THEN statement can be used as an argument to a function.

The exceptions to this rule are all expressions which cause a "shift of location counter", e.g. a GO statement. Since the effect of a GO is to move the execution to another place in the program, there is no way that the (GO label) expression can be thought of as having a value. Similarly, EXIT and RETURN expressions use control to leave the current context, and THROW expressions act like multi-level RETURNS.

The evaluation of the expression starts with the evaluation of the operator, THIS.

Note that the operator is always evaluated, then examined. This differs in several respects from most other LISP systems. First, it is the value of the operator, not its form, that counts. Second, the evaluation does not differ from normal variable evaluation. There are no "function value cells" in YKTLISP. This is a mixed blessing. Unlike other LISP's, simple assignment or binding acts as re-definition, which can be very useful. However, this means that the inadvertent use of an identifier as a variable can cause irreparable damage to the system.

Once the operator's value is found it is examined. If it is the name of "special form" it is treated idiosyncratically, if it is a macro it is expanded and re-evaluated. If it is anything else, it is put aside and the arguments are evaluated, left to right, and the previously obtained operator value is applied.

We will continue the (incorrect) tradition of describing (FOO 1 2) as the application of the function FOO to the arguments 1 and 2, when in reality it is the application of the value of FOO (which could be anything at all) to the arguments 1 and 2. Even in the case of the special forms and the
built in functions this evaluate-first rule holds. (CAR X) is only what it appears as long as the value of the identifier CAR is the (built in function) identifier CAR. If you were to assign the identifier CDR to CAR, (and CAR to CDR), you would cause all future instances of (CAR x) to return the "CDR" of x.

This is somewhat of a lie, in that it is true of interpreted code, but not necessarily of compiled code. The compiler resolves the operators at compile time, and for the special forms and built in functions, "freezes" them, making them insensitive to later redefinitions. This is also true of macro forms, but not of ordinary functions (whether system or user defined).

So, what is a function? A function is usually a LAMBDA expression or a bpi produced by the compilation of a LAMBDA expression. We will ignore the bpi for the moment, as it can be considered as equivalent to its source expression. We could "define" a function, FOO, by assigning a LAMBDA expression to the identifier FOO:

(SETQ FOO "(LAMBDA (X) (COND (X 1) ("T 2")))"

So much for orderly introduction of concepts.

SETQ is LISP's assignment operator. It assigns the value of its second operand to its first operand, which must be an identifier.

COND is LISP's answer to IF ... THEN, and will be dealt with shortly.

" is shorthand for QUOTE, which in turn means "don't evaluate my argument, just use it as is". Notice that only the identifier, T, is QUOTEd. Numbers, such as 1 and 2, may be QUOTEd for neatness, but they are constants and can be entered as is.

If one now evaluates:

(FOO NIL)

the value will be 2. The interpreter, on evaluating FOO and not finding a special form or a macro, evaluated the argument, NIL, which is a constant, and then applied the value of FOO, the LAMBDA expression.

Application of a LAMBDA expression consists of associating the operand(s), (in this context often referred to as arguments) with the bound variable list, in this case (X), and then evaluating the body.

Here we have a single operand, NIL, and a single variable in the bound variable list, X. The effect is to create an environment in which X evaluates to NIL. We speak of this as "binding" the variable X.

The body of this particular LAMBDA expression is:

(COND
 ( (X 1)
 ("T 2")
)

A COND statement consists of one or more clauses, each of which contains a predicate and zero or more expressions. The predicate of each clause is evaluated in turn, until one evaluates to some value other than NIL. When that occurs, the following expressions (if any) are evaluated, and the value of the final one becomes the value of the COND.

In this case, X evaluates to NIL, so the first clause is abandoned. The predicate of the second clause is "T" (or QUOTE T), in its full representation. The value of "T" is T, which is not NIL, so the following expression is evaluated. 2 is a constant, and as the final (and only) expression in the clause, it becomes the value of the COND.

Since the COND is the only expression in the body of the LAMBDA expression, 2 is the value of the LAMBDA expression, and is returned as the value of (FOO NIL).

COND is often referred to as the McCarthy conditional.

29 YKTLISP Program Description and Operations Manual
(COND
  (p1 e1)
  (p2 e2)
  (p3 e3))

is equivalent (in a language in which IF ... THEN can have a value) to:

IF p1 THEN e1
  ELSE IF p2 THEN e2
    ELSE IF p3 THEN e3 ELSE NIL;

Just as a clause in a COND can contain a number of expressions, the last of
which provides the value for the clause as a whole, so the body of a LAMBDA
expression can contain more than one expression. Suppose the value of FOO
were:

(LAMBDA (X Y)
  (SETQ X (TIMES 2 X))
  (SETQ Y (TIMES 3 Y))
  (PLUS X Y))

Then the value of (FOO 5 6) would be 28.

These forms are referred to as implied PROGNS, as they obey the same rules
as the PROGN operator. An expression of the form:

(PROGN e1 e2 ... en)

will evaluate the ei expressions, in order, and return the value of an as
the value of the PROGN expression.

In addition to simple sequential evaluation the full power of branching to
labels is provided. An expression with SEQ as its operator is interpreted
as a collection of expressions and labels, where and identifier in "oper-
and" position in the SEQ is taken as a label, not a variable.

(SEQ
  (SETQ X 5)
  (SETQ Y 1)
  LP
  (COND
    (EQ X 0) (EXIT Y))
  (SETQ Y (TIMES Y 2))
  (SETQ X (PLUS X -1))
  (GO LP))

Will return 32 as its value. The expression (EXIT Y) defines the value of
the SEQ in this case. The value of an SEQ can also be provided by having
execution reach the final expression, which must not be a variable (label)
or a constant. If the final expression is not a GO its value becomes the
value of the SEQ.

The previous example contains assignments to variables, X and Y, which are
not bound. These free uses will search the environment for bindings of X
and Y and used those. This is in general not a good thing to do. Other
LISP's have an operator, PROG, which combines the behavior of our SEQ
with the ability to bind variables. We have chosen to separate these functions
at the most basic level, but we have provided a macro PROG. We can rewrite
the previous example as:

(PROG (X Y)
  (SETQ X 5)
  (SETQ Y 1)
  LP
  (COND
    (EQ X 0) (RETURN Y))
  (SETQ Y (TIMES Y 2))
  (SETQ X (PLUS X -1))
  (GO LP))

Note the change of EXIT to RETURN. The PROG is equivalent to a LAMBDA
expression, and RETURN is the operator which provides a value for a LAMBDA
expression. There is one further difference between these two forms. In
the first (SEQ) example, a GO to a label which did not occur in the imme-
diately enclosing SEQ would not necessarily be an error. It would,
instead, search for an outer SEQ containing the desired label. This search stops when an enclosing LAMBDA expression is encountered. Since PROG is a LAMBDA expression a GO cannot be used to leave a PROG.

The PROG expression has further facilities. Let us modify our example again:

(PROG ((X 5) (Y 1)))
  LP
  (COND
    ((EQ X 0) (RETURN Y)))
  (SETQ Y (TIMES Y 2))
  (SETQ X (PLUS X -1))
  (GO LP))

Here we have specified the initial values for the variables X and Y directly in the declaration list of the PROG.

It must be remembered that in YKLISP there are no unbound variables. Any variable in a PROG without an explicit initial value will be given one of NIL. An attempt to apply a LAMBDA expression to fewer arguments than it has variables will result in an immediate error. A free variable, which has had no value assigned to it will evaluate to itself.

It is hoped that this introduction will allow the reader to follow the detailed descriptions of the various operators which follow.
LISP is noteworthy among programming languages, in that only a rather small kernel of knowledge is required to understand the meaning of its utterances. (The particular dialect YKLISP that is defined here derives from IBM licensed program 5796-PKL.) The whole question of "understanding" can be stated as:

What does the given sequence of characters (on paper or on a display) represent? How does an expression of the following form evaluate?

Understanding utterances of natural languages and most computer languages through deduction is simply out of the question. Understanding through deduction is unproductive when the underlying rules or axioms are not known.

In a sense, the preceding two questions are really one. This is true because, in LISP, an expression is simply another data object. Its "expression-ness" is only an interpretation placed upon it by virtue of the actions of the interpreter and/or compiler being used. Thus understanding of meaning must be approached in two stages, first, what are the data objects, second, what will the interpreter/compiler "do" with any particular object.

The understanding of YKLISP evaluation is possible through the mastery of the following concepts and the aid of a dictionary of primitive operators. Questions about the intent of a program or certain global understandings may not be answered by this process.

In this description we attempt to convey the underlying rules without using much formal notation. In describing syntax however we use a few conventions:

- { and } are used for metalinguistic grouping.
- | is used to separate alternatives.
- [ and ] are used to indicate optionality.
- The ellipsis "..." is used to denote zero or more instances of the preceding object.

YKLISP DATA TYPES

The following is intended to be an intuitive introduction to the various data objects supported by YKLISP. Formal rigor is surrendered in favor of an effort to impart a sufficient operational understanding of these LISP data objects to make the following sections describing the standard LISP functions easier to use. For the programmer, the information presented here should indicate the range of data types available in the YKLISP system and allow him to make some reasonable selections for use in describing his problems.

It is common, when speaking of LISP data objects, to talk about a vector, or an identifier, or perhaps a list cell, when in fact the object being discussed is actually a pointer to that vector, identifier, or cell. This practice is ubiquitous in the LISP community, and will be employed in this manual. Only in cases where it is vitally important to make a distinction will the more cumbersome term "pointer to a vector" be used.

The pointers used by YKLISP are full words (32 bits) and are rich pointers. This means that in addition to a storage address, they contain (in their high-order bytes) a code indicating the type of object they point to. The reason for having these rich pointers, which do consume more storage space than would otherwise be necessary, has to do with efficiency. Many of the frequently occurring LISP operations require arguments of a specified type. Since the result of an operation performed on an invalid type
of argument may actually destroy the LISP system, checking the types of arguments is essential, and this checking may be more efficiently performed if the type code is part of the pointer.

While it doesn't occur very frequently, garbage collection is a very expensive operation because of the quantity of data it processes. Having type codes associated with pointers makes garbage collection more efficient.

To facilitate the process of garbage collection, pointer type codes are classified into two groups -- pointers to stored objects and pointers to non-stored objects. A type code having a high-order one bit indicates a stored object; a high-order zero bit indicates a non-stored object.

This dichotomy is an artifact of the garbage collector and is somewhat misleading for the programmer, as it classifies binary programs as non-stored objects.

Nevertheless, there is a distinction to be made between pointers which contain the address of stored data, and pointers which might be thought of as containing immediate data. In the latter case, the type code in the pointer indicates the value of this data object is stored in the pointer itself, not in some other storage location. For example, small integral numbers are stored as part of a pointer with an appropriate type code, while floating point numbers are always stored in a memory location whose address is part of a pointer with appropriate type code.

The significance of this distinction between immediate data and stored data affects the concepts of sharing and updating. Stored data may be updated, and if it is shared by several structures, the updated data will also be shared (that is, all of the sharing structures are simultaneously updated). Immediate data is intrinsically non-shareable; therefore, in this sense it is not updatable.

**IDENTIFIERS**

An identifier is a stored data object having at least one component, its p-name (print name). This is a string of characters which is recognized by the reader as representing a particular, unique object, and which is used by the printer in displaying the object. Thus, ABC represents an identifier, and always the same identifier, whether read today, tomorrow or next month. The actual representation may contain additional characters, the so-called letterizer character, which is used to mark characters which normally have a syntactic meaning to the reader. An identifier whose p-name is '123' would print as [123 in order that it be distinguishable from the number 123.

There is a special set of identifiers, called GENSYM (generated symbols), which preserve identity only within a single use of READ. GENSYM are printed as %Gn, where n ≤ 2^{24}-1. The reader, on encountering a GENSYM in the input, will replace the value, n, by a new value, m, obtained by incrementing a stored value by one. Any further instances of %Gn in the input will be mapped into %Gm. This mapping is only continued for a single invocation of the reader. If %Gn recurs during a later invocation of the reader it will be mapped into a different m.

GENSYM identifiers have only a p-name. Other identifiers may have a property list as well. The property list is not directly accessible to the user, and does not appear in the print representation of the identifier. It is manipulated by a special group of functions.

There are a number of distinguished sub-classes of non-GENSYM identifiers. These are the FR and MR objects, further described below. In addition certain identifiers are conventionally distinguished, such as characters, those identifiers having one character pnames, and digits, the identifiers with pnames '0' through '9'.

There are currently thirteen MRs in YKTLISP. (A fourteenth, CASE, has been proposed, but not yet implemented.) These are:
As will be reiterated below, these identifiers have built-in meanings.

A second group of identifiers with built-in meanings are the FRs. There are currently forty-three of these, but more may (and certainly should) be added in the future. They are:

<table>
<thead>
<tr>
<th>identifier</th>
<th>function</th>
<th>quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLY</td>
<td>EVAL</td>
<td>MDEFX</td>
</tr>
<tr>
<td>APPLY</td>
<td>EVAL1</td>
<td>MRP</td>
</tr>
<tr>
<td>ATOM</td>
<td>EXI</td>
<td>MSUBRP</td>
</tr>
<tr>
<td>BITSTRINGP</td>
<td>FIXP</td>
<td>NTUPLEP</td>
</tr>
<tr>
<td>CALL</td>
<td>FLOATP</td>
<td>NULL</td>
</tr>
<tr>
<td>CALLX</td>
<td>FRP</td>
<td>NUMBERP</td>
</tr>
<tr>
<td>CAR</td>
<td>GENSYMFP</td>
<td>PAIRP</td>
</tr>
<tr>
<td>CDR</td>
<td>IDENTP</td>
<td>PLEXP</td>
</tr>
<tr>
<td>CLOSURE</td>
<td>LINTP</td>
<td>REALVEC</td>
</tr>
<tr>
<td>CONS</td>
<td>LISTP</td>
<td>REFVECP</td>
</tr>
<tr>
<td>EQ</td>
<td>MDEF</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

**PAIRS**

A pair is a stored data object having two component objects which are referred to as the CAR component and the CDR component (for historical and compatibility reasons). The storage allocation for a pair is two contiguous full-words. Both of these words contain pointers. The CAR component occupies the first word; the CDR component occupies the second word. Since a pointer is used to represent any LISP data object, a pair is an association of two completely arbitrary LISP data objects.

Sometimes it is useful to illustrate LISP data structures. The convention we will use for pairs is a box diagram. Given a pair with a CAR consisting of an arbitrary object, a, and a CDR consisting of an arbitrary object, b, we will draw it as:

```
  *   .   .
     |   >b
     >a
```

Figure 4. Box representation of pairs.

Two basic functions are provided for selecting part of a pair. CAR or CDR applied to a pair returns as its value the corresponding component of the pair.

The print representation of a pair is formally a left parenthesis followed by the print representation of the first element of the pair, a blank, a period, a blank, the print representation of the second element of the pair, and finally a right parenthesis. Or, for the previous example:

```
(a . b)
```

In actual practice cases, however, a simpler or more complex print representation is used.

There is a more compact notation for lists, which is preferentially used, while the existence of shared sub-structure elicits a more complex notation.
Lists

Lists are composite objects created from pairs by applying a conventional interpretation to the pair data type. Thus each pair is a list whose CAR component is interpreted as the first element of that list, and whose CDR component is interpreted as the remainder of that list.

(Note: It is likewise possible to give an interpretation of pairs as trees or rooted directed graphs, however, the use of the list interpretation is assumed by the majority of functions provided by the system.)

The distinguished object NIL is used to denote an empty list. Thus, if the CDR of a pair is NIL, there are no remaining elements in that list.

Having NIL as its CDR component is only one way in which a pair may be the end of a list. If the CDR of a pair is any LISP data object other than a pair, that pair terminates a list.

For the purposes of functions which operate on lists, the CDR component of the pair terminating the list is not considered to be part of the list.

Using the box notation, a list of three elements, A, B and C, would have the following structure:

(A . (B . (C . ())))

Figure 5. Box representation of list, corresponding to dot notation.

Note the convention of representing a pointer to NIL by a / in the appropriate box.

The print representation of a list is a modification of the representation of its component pairs as described above. This modification is intended to improve readability by eliminating some of the parentheses and divulging the sharing of data; however, the inclusion of some (or all) of the deleted parentheses is always acceptable in input data. During printing, when a pair is pointed to from the CDR of another pair, the separating period and blank of the original pair and the enclosing right and left parentheses of the CDR are not printed. In addition, when the terminating pair of a list has NIL as its CDR component, that NIL and the space, period and space which would separate it from the CAR value are not printed. (Note, that if NIL is represented by its alternative form, (), the first rule has the same results automatically. This seems more complicated when described in words than when illustrated by example.

Thus, the list

(A . (B . (C . NIL)))

would appear as

(A B C)
when printed. While
\[(A . (B . (C . D)))\]
will print as
\[(A B C . D)\]
We will reflect this in future box diagrams by presenting them horizontally, thus the previous list will be drawn as:
\[(A B C)\]
\[\begin{array}{c}
    \rightarrow \text{\[.\]} \rightarrow \text{\[.\]} \rightarrow \text{\[/\]} \\
    \rightarrow A \quad \rightarrow B \quad \rightarrow C
\end{array}\]

**Figure 6.** Box representation of list, corresponding to list notation.

Since a pair is a perfectly reasonable element of a list, it is possible to create lists which include themselves, or parts of themselves, as elements. YKTLISP uses a general scheme for input/output which indicates the sharing of data. This sharing scheme, as well as other aspects of the YKTLISP input/output system, makes use of a break character which is defined in the standard system as percent (\%). An input expression written:

\[\%L1=(A . \%L1)\]
generates a pair whose CAR component is a pointer to the identifier \(A\) and whose CDR component is a pointer to the pair itself. The structure is:
\[\%L1=(A . \%L1)\]
\[\begin{array}{c}
    \rightarrow \text{\[.\]} \\
    \rightarrow A
\end{array}\]

**Figure 7.** Box representation of a cyclic list.

The list interpretation of this pair would be a circular list -- effectively an infinite list of \(A\)'s.

This sharing notation need not generate a circular list. For example, the expression:

\[\%L1=(A . \%L1)\]
generates a list containing two elements. The first element is the list containing a single element -- the identifier \(A\) -- and the second element is another identical pointer. This is to be distinguished from the expression:

\[(A) (A)\]

which also generates a list of two elements, each of which is a list containing the single identifier \(A\). In this case, however, the two elements are different pointers, although they point to equal (but separately stored) lists.

YKTLISP syntax and semantics 27
These two structures are:

\[(X1=(A) X1)\]

\[(A) (A)\]

Figure 8. Box representation of equivalent shared and non-shared lists.

For purposes of accessing the elements of the list, both expressions are equivalent (but note that the list having the shared data requires less storage). These two lists are not equivalent with respect to updating. That is, the product of updating one may not be the same as the product achieved by the same updating operation applied to the other.

YKLISP has two primary output operators, PRINT and PRETTYPRINT. PRINT produces a continuous sequence of characters, with all shared structure exposed, whether cyclic or not. PRETTYPRINT produces a formatted representation, with blanks and new-lines inserted where it is deemed appropriate. However, PRETTYPRINT does not expose non-cyclic sharing. Thus the two (different) structures presented here would appear the same when PRETTYPRINTed. PRETTYPRINT has no rules for displaying cyclic structures, and defaults to the unformatted PRINT representation if any are present in its operand.

Note that the system provided read-eval-print supervisor uses PRETTYPRINT for echoing its input and displaying the results of evaluation. This, in turn, means that non-cyclic sharing will not be visible during interactions with the supervisor. LISPEDIT, also, does not explicitly show sharing, cyclic or not, but it does inform the user of its presence.

In general, if it is true of two structure that corresponding accesses yield equivalent values then it can be said that the structures are equivalent trees (see E2UAL function). If it is true that the products of some updating operation applied to two structures would leave them E2UAL, then the structures can be said to be equivalent rooted directed graphs (see UE2UAL function).

**Numbers**

YKLISP operates on three basic types of numbers. A basic numeric data item may be an integer or a real (also called a floating point number, or simply a float). Integers, in turn, are divided into two types, depending upon their value. Integers in the range \(-2^{32}\) to \(2^{32}-1\) (-67,108,864 to
67,188,863), All other integers are represented as large integers. The small integer format stores the numeric value as part of a pointer address field, and so achieves greater efficiency in computation and storage than the large integer format. All integers are stored exactly by LISP. The only limitation on size is the available space in the heap.

Real numbers are stored using System/370 double precision floating point format, yielding 53 to 56 bits of precision for the mantissa and a range of up to (about) 10^{37}.

The print representation for a real number always includes a decimal point to distinguish reals from integer values. This decimal point must be preceded by at least one decimal digit, to avoid possible confusion with the period used in printing pairs. A minus sign may precede the first digit to indicate a negative value.

Both integer and real numbers may be followed by a decimal exponent formed by the latter E, a plus or minus sign (plus is optional), and the exponent magnitude expressed in decimal digits.

There are two parameters which control the way in which real numbers are translated into their print representations for output. FUZZ refers to a value used to define the intended precision of real number operations. Two real numbers, X and Y, are equal in the LISP system if

\[ \|X\| - \|Y\| < FUZZ \times \text{maximum } \{|X|, |Y|\} \]

Insofar as printing a real number, X, is concerned, a character representation is generated for the value in the range

\[ X-FUZZ\times|X| \text{ to } X+FUZZ\times|X| \]

which results in the shortest character string. This print representation may include an exponent, in which case there will be exactly one decimal digit before the decimal point, or in cases where the number of digits (exclusive of decimal point and a possible minus sign) needed to represent the numeric value is less than NDIGITS, no exponent will be printed and the decimal point will be placed wherever is required.

The user may specify values for FUZZ and NDIGITS by using the function SETFUZZ.

VECTORS

YKTLISP vectors may be classified into two general types: pointer vectors and non-pointer vectors. Pointer vectors, as the name implies, may contain references to any LISP data objects (including themselves, so circular structures are possible). Pointer vectors are further classified as reference vectors and selector structures.

Non-pointer vectors contain binary information -- that is, data which cannot contain references to other data objects. Thus, non-pointer vectors are non-descendable from the point of view of the garbage collector and structure-dependent functions such as EQUAL and PRINT. Non-pointer vectors are further classified as bit vectors, character vectors, word vectors and real vectors.

Except for bit vectors, vectors may have any length for which sufficient space exists in the heap. Bit vectors may have a maximum of 2^{24}-1 (16,777,215) elements (bits).

All vectors use zero-origin indexing for referencing their components. The function ELT is a general vector accessing function, applicable to any type of vector with non-zero length. Thus

\[ \text{ELT vector 0) } \]

is always the first element of vector. Of course, if the vector has a length of zero, i.e. contains no components, then any use of ELT (or any other accessing or updating function) is in error. Other accessing functions, tailored to a particular type of vector, are provided because they
are more efficient in execution, or because a more specific check on the type of argument is desired. These are described in the section on vector functions.

Reference, Word and Real Vectors

The print format of a reference vector uses angle brackets to delimit the extent of the vector and blanks to separate elements of the vector:

<COMPO COMPl ... COMPN>

where COMPN is the print representation of the LISP data object referenced as the n'th element of the reference vector. We will occasionally use the a box diagram for a reference vector.

<A B C>

![Box diagram of a reference vector]

Figure 9. Box representation of a reference vector.

The print representation of a word vector is:

%<COMPO COMPl ... COMPN>

where COMPN is a number between $2^{32}-1$ and $-2^{32}$.

The print representation of a real vector is:

%F<COMPO COMPl ... COMPN>

where COMPN is a number in the S/370 double precision floating point range.

Character Vectors

Strings (character and bit vectors) share a special storage characteristic in the YKTLISP system. For reasons of economy (of both storage and processing time) they are stored in contiguous blocks of storage. Nevertheless, because it is considered desirable to allow them to vary in length, a compromise has been achieved which involves maintaining two separate pieces of length information for each string. One length reflects the amount of storage allocated for the string, in terms of the number of elements which may be put into the string without having to allocate more storage for a larger string. The other length refers to the current number of elements which are actually used, which is less than or equal to the capacity of the string.

There are two input/output representations for character vectors. The more general format is:

%&'c...'

where '&k' is the maximum number of characters which could be put into the vector for the character string being read or printed (see the figure depicting string formats below). The actual contents of the character string '&c...' reflects only the current length of the string, and might be
null. Any character may be included as part of a character string; however, the string delimiter character and the letterizer character must be treated specially. In order to avoid confusion about whether a string delimiter character actually delimits a string or is intended as a data character in a string, every occurrence of the string delimiter character as a data character in a string must be prefixed by a letterizer character. This letterizer character is not part of the character string in storage; it is created during output by the print routine, and discarded during input by the read routine. Likewise, every occurrence of the letterizer character as a data character in a character string must be prefixed by the letterizer character. For example, the string

```
''
```

contains one character (a string delimiter), and the string

```
''''
```

contains two characters (a letterizer and a string delimiter).

When it is necessary to represent a character string whose total capacity is not larger than the shortest vector necessary to contain the characters specified, the simpler form:

```
'c...
```

may be used. This designates a character vector which may have zero, one, two or three unused elements. If \( N \) is the number of real characters in a string (letterizing characters are not counted), the number of unused elements for this simplified notation is residue \( (N-1)/4 \).

Example: to specify an eight-element character vector containing the letters F U N C T I O N, write:

```
'FUNCTION'
```

This vector will have space for nine characters and a current length of eight. To specify a vector with a capacity of 100 characters, but with a current length of zero, write:

```
'X100'
```

**Bit Vectors**

The input/output format of bit vectors is similar to the format for character vectors; however, 4-bit segments are represented by one hexadecimal character and the current length field is a count of the number of bits in the vector, not a count of the number of bytes. Only the characters 0...9 and A...F may be specified as part of a bit string.

There are variant input/output representations for bit vectors, depending upon the current length of the vector being considered. For bit vectors whose length is a multiple of four bits, the format is:

```
%Bk'x...
```

where 'k' is the maximum number of bits which the specified vector could contain. The actual contents of the bit string 'x...' reflects only the current length of the string, and might be null.

As with character vectors, the maximum length field is optional and may be omitted when representing a vector of length consistent with the explicitly specified data. A bit vector specified without an explicit maximum length 'k' and with up to 28 unused elements has the format:

```
%B'x...
```

For bit vectors whose current length is not a multiple of four bits, the format is:

```
%Bk:='x...
```
where 'k' is as previously defined and 'c' is the current number of bits in the string. A bit vector specified without a maximum 'k', but with a current length 'c' and with up to 31 unused elements has the format:

`KB:cX...`

**Binary Program Images**

A bpi is a binary program image object which is:

- An mbpi, a machine language macro.
- An fbpi, a machine language function.

A bpi is the product of either the LISP compiler or the LAP assembler. The semantics of applying a bpi that was compiled from a defining expression is similar to the interpreted semantics of applying the expression. The compiler works by transforming the original expression into a new LISP expression, in the process performing macro expansion and dealing with certain operators in special ways. The interpreted semantics of this new expression may differ slightly from those of the original expression. The new expression is then compiled into machine code which has identical semantics with certain exceptions.

These exceptions are of two kinds. First, any program which treats itself as LISP data will fail, as it will have been transformed from a list structure into a bpi. Second, certain operators (specifically RETURN and EXIT) will fail in compiled code if they are the result of a delayed evaluation. I.e., if \( \text{FOO} \), in \( (\text{FOO} 12) \), evaluates to RETURN at execution time, the compiled program will not behave in the same way as the interpreted program.

It is not possible to print binary program images in a form which would permit them to be subsequently read by LISP and used like the original object. There are several reasons for this. The major difficulty being the relationship between the binary program and the entire LISP system, which makes the same program printed at one time from a particular LISP system incompatible with another LISP system, or possibly even with the same LISP system at a different point in time.

Therefore, since it frequently occurs that an object being printed contains references to binary programs (e.g. in a backtrace), a convention is used which incorporates the name of a binary program (that is, the identifier associated with the bpi when it was compiled) in the form:

`%SUBR.bpiname` or `%MSUBR.bpiname`

where SUBR is used for functions with evaluated arguments, and MSUBR is used for macros (functions with unevaluated arguments).

If an attempt is made to read such a form, the read program will emit an error message and use the .NOVAL object instead of a binary program.

**Funargs**

A funarg is an expression closure — that is, the combination of an expression with a specific environment in which that expression is to be executed. It is represented as a pair-like object:

`%FUNARG.(expression . sd)`

where the first element is the actual expression, which will evaluate to an object which may be applied) and the second element is a state descriptor which defines the environment.

WARNING: while it is possible to extract the components of a funarg, the user is strongly advised to refrain from this practice. When the compiler encounters a LAMBDA expression as an argument, rather than as an operator, it generates code to create a funarg, with a compiled version of the LAMBDA expression as its expression part. These binary programs are com-
piled with an understanding of their immediate environment, including the absolute offsets needed to access variables in the stack frame captured by the sd component of the funarg. By executing such a BPI in another environment, unpredictable action (including failure of the YKLISP system) may occur.

STATE DESCRIPTOR

A state descriptor is an elementary data object generated by the STATE basic macro. It is conceptually a pointer to a particular stack frame, which serves to define either an environment (a set of identifier-value associations) or a previous state, which denotes a specific point in the application of a FUNARG. Practically, state descriptors have the capacity to contain some control information, since this is required by the garbage collector and by their use to determine validity of shallow bindings. Thus they are five-word objects and are processed only by a limited set of functions which are prepared to maintain their structure.

Creation of a state descriptor ensures that the related stack frame will be retained until the state descriptor is deleted by the garbage collector when there are no references to it.

State descriptors serve two purposes. First, they define an environment which may be used to create function closures. Second, they are actually saved states which may applied in order to affect a transfer from the current state to the saved state. Execution will subsequently proceed in the environment of the saved state, at the point immediately following the STATE operation which created the saved state. When a state descriptor is applied, it must have an argument, which is evaluated in the environment initiating the application. The value resulting from this evaluation becomes the value of STATE when execution resumes in the saved state.

A state descriptor is essentially unprintable, in that it contains the state of the machine as it existed at the time of its creation. If one is encountered by the print function it is represented as

\[ \text{x.SD.xxxxxxxx} \]

where "xxxxxxxx" is the hexadecimal representation of the actual pointer.

STREAMS

Streams, like lists, are not primitive data objects. They are an interpretation of composite data structures consisting of pairs and vectors. There are two distinct types of streams, each with its own set of operators.

The streams which are managed by READ, PRINT and their related operators can be viewed as producers or receivers of characters. These are the streams which allow YKLISP to communicate with the console, and to read and write files which can be edited by the user.

The streams which are managed by RREAD, RWRITE and their related operators can be viewed as producers or receivers of arbitrary LISP data objects. The PADS files which correspond to these streams are, in general, not readable by the user.

All operations on streams are updating. That is, an operation which changes the value of, e.g., the CAR of a stream does so by a REPLAC operation. This insures that all processes which have access to a particular stream remain in synchronization.

Character streams

All character streams are pairs, with the CAR of the stream being the current character, or an "end-of-line" flag.

YKLISP syntax and semantics 33
The simplest form of a character stream is a list of characters:

\[(A \ B \ C)\]

\[\rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow /\]

\[\rightarrow A \rightarrow B \rightarrow C\]

Figure 10. A list, interpreted as a character stream.

The primitive operations on such a stream are NEXT and WRITE.

After an application of NEXT

\[(B \ C)\]

\[\rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow /\]

\[\rightarrow A \rightarrow B \rightarrow C\]

After an application of WRITE, with X as first operand

\[(X \ A \ B \ C)\]

\[\rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow 1\]

\[\rightarrow X \rightarrow B \rightarrow C \rightarrow A\]

\[1(\text{Note, only the pair marked by } 1 \text{ is newly created.})\]

Figure 11. Effect of NEXT and WRITE on stream, Figure 10.

One peculiarity of the stream interpretation of lists should be noted, that the NEXT and WRITE are mirror operations. If a list is produced by a series of WRITEx, the repeated application of NEXT will produce the characters in the reversed order.

The form of character stream more usually employed is called a fast stream. This is a list, usually of a single element, the final CAR of which is a reference vector containing the information required for performing I/O to or from a DASD device or a terminal.

Fast streams contain a number of component which may be accessed or changed by various operators. These include an association list, with entries specifying the direction of the stream (INPUT or OUTPUT), the device type, the file name, etc., as needed. There is a buffer, containing the current line from the device, with an index designating the current character in the buffer. A stream specific function defines the action to be taken when the buffer is to be disposed of, either refreshed from an input device, or written to an output device.

Two special configurations of a fast stream indicate end-of-line and end-of-file conditions, these may be tested with the predicates EOLP and EOFP.
Key addressed streams

The second type of stream provided by YKTLISP supports key addressed, random access, files. These will be referred to as libraries. Each member in such a file represents a single LISP data object. Associated with each item is a key and a class designation. The key is either a string or a GENSYM identifier, while the class is a small integer in the range 0 (zero) to 255.

The usual file type for such files is LISPLIB.

Unlike character streams, libraries may contain an external representation of a BPI, and are often used as an analogue to VM TXTLIB files. It must be noted that an existing BPI can not be written to a library, but can only be placed in one by the assembler.

The basic operators which deal with libraries are distinguished by a prefixed R on their names, for random access. In addition there exist a set of operators, including LOADVOL, SUBLOAD etc, which manage these files and allow operator definitions to be loaded from them.
EXPRESSIONS: FUNCTIONS, MACROS AND SPECIAL FORMS

The syntax of YKTLISP is described in the well known list form. While the YKTLISP kernel is thought to exhibit an improved structure over many other LISP dialects, it undoubtedly has some weaknesses. For instance, certain identifiers have reserved meanings when applied. The readers' criticism is invited.

**EXPRESSONS**

The primitive expression classes are constants, variables and lists.

A LISP expression is one of:

- $c$, denoting a constant,
- $id$, an identifier, denoting a variable,
- $(operator \ [operand \ ...\ ])$, a list, denoting an operator-operands combination where the operator and each operand are expressions.

**CONSTANTS**

The evaluation of constants is trivial: constants are idempotent, i.e. they evaluate to themselves.

YKTLISP has the following broad classes of constants:

- decimal-numbers (integer and real)
- applicative-constants (sds, bpios and funargs)
- NIL
- ranked-arrays (vectors and strings)

The properties of these constants have been described in "YKTLISP data types" on page 23.

**VARIABLES**

Variables are identifiers which have had values associated with them.

The rules for evaluation variables are described in "Bindings and variable referencing" on page 41.

**LISTS AS EXPRESSIONS**

The third form that an expression can take is a list. These lists are of the form $(operator \ [operand \ ...\ ])$.

1. Special forms: a small number of primitive expressions, each with its own rules. The operators are referred to as mrs.

2. Macro-expressions: expressions whose operators receive the expression (of which they are the operator) as their argument, and whose value is evaluated again. The operators are either MLAMBDA expressions, mbios or macro funargs.

3. Functions: expressions whose operators receive the values of the operands, and whose value is that returned by the operator. The operators are LAMBDA expressions, bpios, functional funargs or built-in operators, referred to as frs.
4. Structure accessing expressions: expressions whose operators are macros defined by the DEFINE-STRUCTURE operator. Used to access components of user defined structures.

5. Jaunt statements: expressions whose operators are state descriptors, and which cause the resumption of a previously saved environment.

If the value of the operator is not a member of one of these classes, it is set aside and the operands are evaluated. It is then retrieved and repeatedly reevaluated until its value is an applicable object, in which case it is applied, or a macro operator, constant or idempotent, non-applicable object, in which case an error break is taken.

Thus, the type of application depends on the value of the operator. It could be considered unfortunate that each type of application is not represented by a distinct syntax. The resulting lack of transparency is balanced by the flexibility of the delayed interpretation that can be considered a feature of this LISP. Indeed the lack of distinction makes the definition of most operators a free choice between macro definition and ordinary function definition.
At its simplest, evaluation in YKTLISP consists of accessing and updating the environment (see the next section).

If an expression is a constant it evaluates to itself. If it is a variable, the environment of evaluation is searched for its current value. If it is a list, its operator is evaluated and the expression is classified.

In fact, the case of

\((\text{exp} \text{ list} [\text{exp} ...]) [\text{exp} ...]\)

where \text{exp} evaluates to \text{LAMBDA}, is specially recognized. Semantically, it need not be, as the operator would evaluate to a funarg which captures the environment, but in order to avoid the creation of a state descriptor, we note the explicit presence of the \text{LAMBDA} expression, and treat it as a special case when applying it.

The classes of composite expressions are: special forms, functions, functional funargs, macros, macro funargs and jaunts.

**OPERATOR — special form**

\((\text{OPERATOR} [\text{operand} ...])\)

Each special form has its own application rules. Special forms differ from macros (below) in that they are built into the system. The compiler also understands them, and treats each separately.

**OPERATOR — built in function**

\((\text{OPERATOR} [\text{operand} ...])\)

A built in function operates directly on its operands. It has no variables, and no stack frame is created for it. It, itself, may cause a stack frame to be created (e.g. CALL, APPLY, et cetera). The compiler treats many of the built in functions specially, producing in line code for them.

**OPERATOR — function**

\((\text{OPERATOR} [\text{operand} ...])\)

A function is a \text{LAMBDA} expression or a \text{fbpi}.

A function contains two components, a list of variables and one or more expressions. (\text{LAMBDA} expressions contain these explicitly, \text{bpi}’s, implicitly.) When a function is applied to the values of its operands those values are paired with the variables of the function, and used to augment the environment. At the same time the control environment at the point of application is saved and the environment of execution is augmented to evaluate the expressions of the function. These expressions are evaluated in this new environment. When the evaluation completes a value is returned (by reverting to the previous state of the environment) which becomes the value of the expression.

More details as to the changes to the environment are given in the following section.

**OPERATOR — function + compiler macro**

\((\text{OPERATOR} [\text{operand} ...])\)

A number of functions have corresponding macro definitions, available to the compiler. These allow the production of in line code in \text{bpi}’s, by providing the compiler with code generators. These macro definitions are not generally available to the interpreter.
The functional definitions of these operators, which are available to the interpreter, behave as in the previous case.

**OPERATOR — MACRO**

(OPERATOR [operand ...])

Macro application differs from functional application in two ways.

First, the items in the expression are not evaluated, and the variables of the macro are associated with the entire expression as it was before the operator was evaluated. See the description of MLAMBDA, page 53.

Second, the value returned by the application does not become the value of the expression. Rather, it is treated as if it had occurred in the place of the original expression, and is evaluated in turn.

**OPERATOR — funary**

(OPERATOR [operand ...])

The expression component of funary is applied, as a function or as a macro, depending what it is. The application takes place in the environment of evaluation which is captured by the state descriptor component of funary.

**OPERATOR — structure**

(OPERATOR access-path instance)

Where structure evaluates to a macro created by the DEFINE-STRUCTURE operator. Returns the component of instance described by access-path.

**OPERATOR — jaunt**

(OPERATOR item)

The act of applying a state descriptor (sd) to an operand is referred to as jaunting.

A state descriptor binds no variables, and it does no explicit computation. It causes execution to resume at the point where sd was created. This is an application of the built in function STATE.

The program acts as if the original application to the function STATE had just completed. However, the value returned is item, rather than the sd originally created. See page undefined.

**THE ENVIRONMENT OF EVALUATION AND EXECUTION.**

All evaluations in YKTLISP take place with respect to two environments, the environment of execution and the environment of evaluation.

Historically, these have been referred to as the control and the environment. This terminology has the virtue of brevity, and will at times be used. In what follows, read "environment" (with no prepositional phrase) as "environment of evaluation", and "control" as "environment of execution".

The operators which affect the environment of execution provide the control structures for the language. The operators which affect the environment of evaluation include the assignment and evaluation operators.

The environment of evaluation consists of an ordered set of variable bindings, that is, locations which contain values, each with an associated variable (an identifier).
The environment of execution consists of the information needed to resume suspended evaluations, whether the suspension is due to a simple application, or a state saving operation.

**Stack frames**

The two environments are embodied in a single structure, the stack, with an extension, the global environment, which is part of the environment of evaluation. The stack, in turn, is composed of frames. A frame is created for each application of a LAMBDA or MLAMBDA expression or bpl and for each invocation of the interpreter (via EVAL or EVAL).

Note that evaluating a PROG is equivalent to the application of a LAMBDA expression. Since many of the operators used for control purposes are macros which generate LAMBDA expressions, it is not always apparent when a new stack frame will be created.

Each stack frame contains:
- a (possibly empty) set of variable bindings, consisting of value cells and a display which contains the variable names and their FLUID/LEXICAL status.
- a flag, indicating whether LEXICAL variables should be ignored from this point on, if the search for a binding reaches this frame.
- a pointer to the next frame to be examined for a variable's value, if it is not found in this frame. (The environment chain. This may differ from the next item, as will be detailed below.)
- a pointer to the frame from which this frame was created, and to which control will return when the evaluation in this frame completes. (The control chain.)
- A flag marking this frame as a catch point, if it is one.
- the necessary information to allow evaluation to resume when control returns to this frame.
- miscellaneous "housekeeping" information to allow the proper maintenance of shallow binding cells for FLUID variables.
- other information, accessible only to assembler language programs.

The operation RETURN is seen as the abandonment of the current stack frame and the resumption of evaluation in the previous frame on the control chain, with a value provided. Application of a LAMBDA expression is the creation of a new stack frame and the initiation of evaluation in it. THROW consists of following the control chain, performing housekeeping operation as each frame is abandoned, and resumption of evaluation in a frame marked as a catch point.

This model is too incomplete to account for SEQ, EXIT and GO. They make use of information which is hidden from any user program.

**Bindings and variable referencing**

A reference to a variable must be resolved to a particular binding. The binding, once found, can be either accessed or updating. Accessing a binding is equivalent to evaluating the corresponding variable. Updating a binding is equivalent to assigning a new value to the corresponding variable. The search commences in the current stack frame, looking at all variables. Once it passes a "closed" contour, that is a frame corresponding to a non-lexically present LAMBDA (see section on Scope of variables) only FLUIDly bound variables are looked at. If the variable is not found in the stack the current global A-list is searched. If it is not found and if the A-list terminates in an SD the global A-list associated with that SD is searched, and so on until the variable is found or a non-SD terminator is reached.
(Note that this search is often avoided by the use of shallow binding cells. In practice, once the search has left the immediate (lexical) scope of the variable a "look aside" is performed. If the variable has a shallow binding cell (not all do) and if that cell is current (that is it was last set in the current environment) then it is used to access the current binding. If it is not current the search is carried out and the shallow binding cell is "refreshed", to avoid further searches, until the environment is changed again.)

The frame searched during this process are normally the same as the "control chain", that is the chain of stack frames from called function to calling function. However, whenever a function is applied or EVAL, APPLY or MDEF are called a fork occurs, and the environment chain following the stack frames captured by the sd involved. In addition, in each of these cases the global A-list in the SD becomes the current global environment.

THE ENVIRONMENT OF COMPILATION.

In YKTLISP there exists another environment which must be understood. This is the environment of compilation.

Compilation involves a partial interpretation of the function being compiled. In particular, all macro operators must be macro-applied, that is expanded, at the time of compilation. The resulting, macro free, lisp code is then translated into machine code.

YKTLISP provides two special forms, XCODE and FRXCODE, which allow the programmer, either directly or indirectly via macro definitions, to provide machine code sequences to the compiler for expressions. Many of the basic operators, particularly the so-called "Q" operators (see "A note on naming conventions" on page 45) are implemented using this feature.

Macro definitions are not applicable however. Thus it is desirable that the number of macro definitions in the system be kept to a minimum. To escape this dilemma the systems contains dual definitions of approximately 200 operators. In the "normal" environment, that is the environment in which user programs are evaluated, these operators have functional definitions. In addition macro definitions exist, macros which expand to XCODE or FRXCODE expressions. These macro definitions are bound in a special environment, the operator recognition environment (reduced to OR as a component of certain operator names. See "Operator definition" on page 129). The OR environment inherits the normal global environment, but not the stack portion of the current environment of evaluation. Thus, any operator not bound explicitly in the OR environment will have as its value that that the user normally sees.

The compiler evaluates all identifiers occurring as operators in the OR environment.

The definition facility binds a number of FLUID variables, as may the program from which it is invoked. The OR environment does not contain these bindings, thus no conflict can arise between the system's FLUID variables and the operator values in the OR environment.

In order to prevent similar conflicts between operators used during macro expansion and system FLUID variables the expansion itself is carried out in a second special environment, the macro expansion environment, referred to as the MA environment. The MA environment, like the OR environment, does not contain any of the users or systems FLUID bindings. Unlike the OR environment it normally contains no bindings of its own, simply be equivalent to the systems global environment.

Various functions exist to allow the user to add new definition to either of these environment, permanently or temporarily. See "Operator definition" on page 129.

42 YKTLISP Program Description and Operations Manual
The following descriptions consist of two parts. In each subsection there is a short discussion of the operators which are grouped together. This discussion may be split, with an introduction at the beginning, and more detailed comments at the end. Following the introduction will be a description of each function in the group. The descriptions will follow a stereotyped format, with one or more examples followed by text, thus:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>operator type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(OPERATOR operand1 operand2 [operand3 ...])</td>
<td></td>
</tr>
</tbody>
</table>

Upon receiving its operands operator merges them in an order determined by the presence or absence of a fourth operand. If all supplied operands are not of sufficient strength an error break is taken.

Figure 12. Description of a fictitious operator.

There are also variants on this form, with variables and key words rather than operators.

In addition to the cases set forth on page 39, entries of the following forms will be found.

WORD — key word

An identifier which must occur explicitly, either as an argument to some operator, or as a component of a structure. For example, the property names of the elements of OPTIONLIST, see page undefined.

IDENTIFIER — variable

An identifier which contains a system provided initial value. It may be used as a free variable by one or more system operators, in which case its value will affect the behavior of those operators in some way, for example, CUROUTSTREAM. It may contain a value of use to user programs, for example STACKLIFO.

IDENTIFIER — system command

(CALLBELOW 'IDENTIFIER' [operand ...])

A string which is interpreted by the system dependent (VM, TSO, etc.) module as a request for service.

Note that only the VM system dependent module has been implemented to date.

The operands depend on the specific command, but must be strings, word vectors or small integers.

PHRASE — concept

A discussion of some concept felt to be important. These will often reprise information from the syntax and semantics section at a point where it is relevant.

Arguments will be shown highlighted, as in the example, and their names will usually be chosen to reflect the type of value expected. In order to minimize the length of the exemplar line, the types will be abbreviated, as shown in Figure 13 on page 44.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>object of any type</td>
</tr>
<tr>
<td>pair</td>
<td>pair</td>
</tr>
<tr>
<td>list</td>
<td>pair, interpreted as a list</td>
</tr>
<tr>
<td>strm</td>
<td>stream</td>
</tr>
<tr>
<td>rstrm</td>
<td>random access file</td>
</tr>
<tr>
<td>vec</td>
<td>vector (including string)</td>
</tr>
<tr>
<td>str</td>
<td>string</td>
</tr>
<tr>
<td>c-str</td>
<td>character string</td>
</tr>
<tr>
<td>b-str</td>
<td>bit string</td>
</tr>
<tr>
<td>id</td>
<td>identifier</td>
</tr>
<tr>
<td>char</td>
<td>single character identifier</td>
</tr>
<tr>
<td>num</td>
<td>number</td>
</tr>
<tr>
<td>s-int</td>
<td>small integer</td>
</tr>
<tr>
<td>fx-num</td>
<td>fixed number (integer)</td>
</tr>
<tr>
<td>flt</td>
<td>floating point number</td>
</tr>
<tr>
<td>app-ob</td>
<td>applicable object</td>
</tr>
<tr>
<td>exp</td>
<td>expression</td>
</tr>
<tr>
<td>sd</td>
<td>state descriptor</td>
</tr>
<tr>
<td>label</td>
<td>identifier, interpreted as a label</td>
</tr>
<tr>
<td>bv-list</td>
<td>bound variable list</td>
</tr>
<tr>
<td>a-list</td>
<td>association list</td>
</tr>
<tr>
<td>sysdep-area</td>
<td>(str</td>
</tr>
<tr>
<td>file-name</td>
<td>[[id1 [id2 [id3]]]</td>
</tr>
<tr>
<td>file</td>
<td>[(id1 [id2 [id3]]</td>
</tr>
</tbody>
</table>

Figure 13. Abbreviations for operand data types.

If examples are given, they will be in the form of a "trace" of interactions with the normal supervisor, with echoing turned off.

\[
\begin{align*}
X & = 123 \\
(PLUS X X) & = 246 \\
\end{align*}
\]
A NOTE ON NAMING CONVENTIONS

LISP has always had a few conventions for naming functions with common attributes. Most "true/false" functions (predicates) have names ending in P, for example, while many functions which modify their arguments have names beginning with RPL (for replace) or N (origin unknown). Examples are RPLSTRING and REVERSE as contrasted with REVERSE. YKLISP has several conventions of its own (as with the ...P we are not 100% consistent in following them).

You will quickly notice many functions with commas imbedded in their names. These all were originally thought of as "system" functions, which the casual user would not be interested in. As the comma has no special syntactic meaning in LISP370/YKLISP, but acted as a list separator in the previous LISP available at Yorktown this was felt to be a protection from name clashes.

A few functions have names which start with a comma. These are usual "under cover" versions of normal functions. Thus ,PLUS is a two argument generic addition function, used by PLUS which is the multiargument generic addition function.

Many other functions have names prefixed by one or more letters and a comma. The prefix groups the functions, e.g. parts of the compiler start with C, parts of the LISP assembler with L, and general system functions with S.

Many other functions will be seen which start with a Q. This stands for "QUICK", and can have one of two meanings. In all cases it implies the existence of a macro in the compile environment which produces in-line code. In many cases it also implies a lack of type checking.

So QCAR and QCADR do not test their arguments for PAIRness, they assume correct type and act accordingly. The compiler is, at present, not smart enough to elide the type checks in code such as

(COND ((PAIRP X) (CAR X)) ("T X))

so the careful programmer is allowed to circumvent the built in checks by writing

(COND ((PAIRP X) (QCAR X)) ("T X))

Other of the Q operators do type checking, but are quick in that they result in in-line code. These include QMEMQ and QASSQ.

Another group of Q operators are the QS... operators. These (QSPLUS, QSADD1, etc.) assume that their arguments and values are small integers (numbers between -(2^26) and 2^26-1). General they do arithmetic modulo 2^26, forcing correct small integer type codes on their results. They are also aware of each other, and will skip the forcing of type codes on some intermediate results when nested.

Note that there are a few functions starting with Q which are neither in line nor unchecked. These include QSORT and QUOTIENT.

Functions whose names end in Q are usually version of other functions which use EQ rather than EQUAL. Such pairs include MEMBER/MEMQ, ASSOC/ASSQ, UNION/UNIONQ.
ENVIRONMENT OF EXECUTION

These operators do not compute values, per se, but rather provide the control structure of programs. Many of them use their arguments as written, rather than as evaluated. Even when arguments are evaluated, the order of evaluation may differ from the normal, left to right, order of function application.

Just as the operators described in the following section can be thought of as accessing and updating the environment of evaluation, so these operators access and update the environment of execution. (See "The environment of evaluation and execution" on page 48.)

The distinction is by no means clear cut, however. For example, the application of a LAMBDA expression results in an augmentation of both parts of the environment, in that a new stack frame is created, which contains both variable bindings (environment of evaluation) and a control chain pointer which RETURN may use (environment of execution).

Despite the importance of binding on the environment of evaluation it was felt that the control aspect of LAMBDA warranted its inclusion in this section.

SPECIFICATION OF VALUES

Many of the data objects in YKTLISP are constants, that is they evaluate to themselves. This is true of all numbers, strings and vectors, for example. Lists (i.e. pairs) and identifiers, however, generally do not have this property.

If, in an expression, you wish an identifier or a pair itself, rather than its value, you must QUOTE it.

(Note the convention of spelling YKTLISP operators in uppercase when using them as verbs. We will often write QUOTEd or FLOATing when we are speaking about the use of the operators QUOTE and FLOAT, for example.)

It is also possible (in compiled programs) to specify a constant as the value of an expression (evaluated during the compilation process). This is done by using the CONSTANT operator.

QUOTE — special form

(QUOTE item)

The value of a QUOTE expression is just the operand, item. This allows one to mention an identifier, or a list without having it evaluated as an expression. The reader in YKTLISP accepts the form "item as equivalent to (QUOTE item).

(QUOTE X)
Value = X
"ABC
VALUE = ABC

CONSTANT — macro

(CONSTANT exp)

The CONSTANT operator evaluates exp and returns its value.

During compilation, CONSTANT is defined as a macros, which replaces itself by the QUOTEd value of exp. This allows the use of the values of expressions (as evaluated at compile time) as constants.

The bpi resulting from the compilation of

(LAMBDA () (CONSTANT (EXP 3)))

Environment of execution 47
SEQUENCE OF EVALUATION

YKTLISP provides a number of ways of specifying the sequence of evaluation of the expressions which comprise a program.

The most basic is the order of evaluation of the elements of a list, when it is interpreted as an expression. First the operator is evaluated, then, if it is not a macro or special form, the operands are evaluated in strict left to right sequence, and then the operator is applied to them.

If the operator is a macro, application follows immediately, and if it is a special form its own rule holds.

Next, in primitiveness, is the application of LAMBDA expressions, or their compiled surrogates, bpi's. Each such application results in a transfer of control to the body of the LAMBDA expression, its evaluation, and a return of control to the point of application.

Within the body of a LAMBDA expression, as well as in certain other contexts, the implied PROGN rule holds. This entails the left to right order of evaluation, but in this case all but the final resulting value are discarded.

The counterpart of IF ... THEN ... ELSE ... is provided by a number of operators, primarily COND. Here a series of tests are made, and the first to succeed specifies a particular sequence of expressions to be evaluated.

YKTLISP provides a GO operator, which within limited context allows arbitrary sequence of control.

RETURN, EXIT and THROW (with various operators built on them) allow the premature termination of the normal control flow, with control reverting to some surrounding context.

Finally, the application of a state descriptor results in control resuming at the point at which that state descriptor was created. This allows co-routining and back-tracking.

PROGN — special form

(PROGN [...])

The sequential evaluator. The expressions, exp1 ..., are evaluated in order, and the value of the final expression is normally the value of the PROGN expression. If control leaves the PROGN expression, via a GO or RETURN for example, then, in some sense, the PROGN expression has no value.

(PROGN (SETQ X "(A B)") X)  
Value = (A B)  
X  
Value = (A B)  
(PROGN (SETQ X "10") (SETQ Y "20") (PLUS X Y))  
Value = 30  
X  
Value = 10

PROG1 — macro

(PROG1 exp1 ...)

The PROG1 operator evaluates its operands in left to right sequence, in the same way as PROGN, but returns the value of its first exp as its value.

PROG2 — macro

(PROG2 exp exp ...)

48 YKTLISP Program Description and Operations Manual
The PROG2 operator evaluates its operands in left to right sequence, in the same way as PROGN, but returns the value of its second EXP as its value.

SEQ — special form

(SEQ [exp ...])

SEQ defines the scope for labels. Any EXP which is an identifier is not evaluated, but instead acts as a label, for GO statements. The other EXPs are sequentially evaluated, as in PROGN, except when a GO is executed. A GO to a label in the SEQ causes the sequential evaluation to recommence at the following expression.

The value of an SEQ is determined in one of two ways.

If the sequential evaluation reaches the last expression in the SEQ, and if that is an expression with a value (i.e., not a GO, RETURN, etc.), then the value of that expression becomes the value of the SEQ expression.

Labels are treated as if they had a value of NIL. Thus, if the final expression in an SEQ is a label, the value of the SEQ will be NIL, whether control reaches the end via a GO or by normal sequential evaluation.

Alternatively, if an EXIT expression is evaluated inside an SEQ the value of its argument becomes the value of the SEQ.

GO — special form

(GO id)

The GO searches for an enclosing SEQ expression. If it encounters an enclosing LAMBDA or MLAMBDA expression an error break is taken.

If it finds an SEQ it searches it for an instance of id, and if it finds it cause evaluation to start at the following expression. If there is no instance of id the search for an enclosing SEQ is repeated, outside the previously found SEQ.

EXIT — built in operator

(EXIT exp)

The EXIT operator causes control to leave the nearest enclosing SEQ or LAMBDA expression. The value of EXP becomes the value of the SEQ or of the application of the LAMBDA expression.

EXIT is provided to allow the termination of a SEQ expression, with a value. EXIT expressions may occur anywhere within the SEQ.

RETURN — built in operator

(RETURN exp)

The RETURN operator causes control to leave the nearest surrounding LAMBDA or MLAMBDA application, with EXP becoming the value of the application.

Surrounding SEQ expressions are not seen by RETURN, and their evaluation is terminated.

PROG — macro

(PROG list [{exp | id} ...])

The PROG operator combines certain aspects of SEQ and LAMBDA expressions.

It establishes a context of GOs and labels, as SEQ does, and also establishes a scope for variables, as LAMBDA does.
Since PROG is equivalent to a LAMBDA expression (in fact it is a LAMBDA expression with a SEQ expression as its body), GO is not allowed to leave a surrounding PROG. Either RETURN or EXIT will cause an immediately surrounding PROG to terminate, and will provide the value for the terminated evaluation.

Unlike either SEQ or LAMBDA, the value of a PROG which terminates due to the evaluation of its final expression (if it is not a GO, EXIT or RETURN), is NIL.

The sequence of \((\exp | \text{id})\)'s is interpreted in the same way that the body of a SEQ expression is, that is \text{id}'s are not evaluated, but act as labels for GO expressions.

\text{list} is a list of variable which are bound by the PROG. The elements of the list may have one of three forms:

1. identifier, a lexical variable, with an initial value of NIL.
2. \((\text{FLUID} | \text{LEX}) \text{identifier}\), a explicitly FLUID or LEXical variable, with an initial value of NIL. (The form \((\text{LEX} \text{identifier})\) is only required for the identifiers FLUID or LEX.)
3. \((\text{variable} \exp)\), where variable may be either of the two preceding forms, a variable with initial value \exp.

It is important to note that \exp is evaluated outside the binding scope of the PROG. Thus sequences of evaluations such as

\[
\begin{align*}
(\text{SETQ} \ X \ "(2 \ 1)) \\
\text{Value} = (2 \ 1) \\
(\text{PROG} \ ((X \ \text{CONS} \ 3 \ X) \ (\text{PRINT} \ X) \ (\text{RETURN} \ (\text{CONS} \ 4 \ X))) \) \\
\text{Value} = (4 \ 3 \ 2 \ 1) \\
X \ &= \ (2 \ 1)
\end{align*}
\]

where a value of a variable outside the PROG is used, manipulated, but may be unaffected after the PROG completes. Of course, if updating operation had been applied within the PROG, their effects could have been visible afterwards.

**CONDITIONAL EVALUATION**

**COND** -- special form

\((\text{COND} \ [\text{clause} \ldots])\)

This is the IF ... THEN ... ELSE of LISP. Each clause is a list of expressions of the form

\((\text{predicate} \ [\exp \ldots])\)

(Any clause which is not a list is treated as a comment.)

The clauses are examined in order, and the predicate is evaluated. If the value is NIL the next clause is examined.

If the value is not NIL, the remainder of the clause is examined. If there are no \exp\'s the value of the predicate is the value of the COND expression. Otherwise the list of \exp\'s is evaluated as if it were prefaced with PROGN. The value of the final \exp becomes the value of the COND.

In either case, no further predicates are evaluated.

**CASEGO** -- macro

\((\text{CASEGO} \ \exp \ \text{list} \ldots)\)
Where each list is of the form

\[
\text{ITEM \ id}
\]

This operator evaluates \text{EXP} and compares the resulting value with the items from successive lists. These items are not evaluated. If one is found which is EQ to the value of \text{EXP} a GO is performed to the corresponding \text{id}, interpreted as a label.

\begin{verbatim}
(CASEGO 
   (CAR X) 
   (A L1) 
   (B L2) 
   (3 L2) 
   (XYZ D))
\end{verbatim}

Will execute a GO to the label L1 if the CAR of the value of X is the identifier A, to the label L2 if it is the identifier B or the number 3, to the label D if it is the identifier XYZ. If it is none of those four, control will continue following the CASEGO expression.

If a CASEGO expression is used as an operand, and no GO is performed, its value is the value of \text{EXP}.

\textbf{OR — macro}

\begin{verbatim}
(OR [EXP ...])
\end{verbatim}

This operator evaluates the \text{EXPS} from left to right, until the first \text{EXP} whose value is non-NIL. The value of the expression is that non-NIL value, if such exists, and is NIL otherwise.

\begin{verbatim}
(OR exp1 exp2 ... exp3.)
\end{verbatim}

is equivalent to

\begin{verbatim}
(COND (exp1) (exp2) ... (expn))
\end{verbatim}

(OR) has a value of NIL.

\textbf{AND — macro}

\begin{verbatim}
(AND [EXP ...])
\end{verbatim}

This operator evaluates the \text{EXPS} from left to right, until the first \text{EXP} whose value is NIL. The value of the expression is the value of the final \text{EXP}, or NIL, if an earlier \text{EXP} resulted in that value.

\begin{verbatim}
(AND exp1 exp2 expression3)
\end{verbatim}

is equivalent to

\begin{verbatim}
(COND (exp1 (COND (exp2 expression3))))
\end{verbatim}

(AND) has a value of NIL.

\textbf{SELECT — macro}

\begin{verbatim}
(SELECT exp1 list ... exp2)
\end{verbatim}

Where the lists are of the form

\begin{verbatim}
(exp exp ...)
\end{verbatim}

The SELECT operator evaluates \text{EXP1}. It then evaluates the first \text{EXP} of each list, until it finds one whose value is EQ to the value of \text{EXP1}. If it finds such a one, it evaluates the remaining \text{EXPS} in that list as an implied PROGN, returning the value of the final \text{EXP} as the value of the SELECT expression.

If no such list is found \text{EXP2} is evaluated and its value becomes that of the SELECT expression.
SELECT is a macro which generates a LAMBDA expression, so
RETURNs and EXITs in any of the evaluated expressions will sim-
ply return control from the SELECT. In addition, GOs are ille-
gal, as there is no SEQ in the resulting LAMBDA expression.

FUNCTION AND MACRO DEFINITION, VARIABLE BINDING

LAMBDA — special form

(LAMBDA bv-list [exp ...])

The behavior of a LAMBDA expression varies, depending upon
whether it is itself an operator in another expression, or, if it
is an operand.

A LAMBDA expression in operator position is immediately applied
to its arguments. The application is performed by binding the
values of the arguments to the variable declared in the
bound-variable list of the LAMBDA expression. This binding
process can be conceptualize as follows. (The actual process
differs somewhat, but the effect is the same.)

Given a LAMBDA expression (LAMBDA bv . body) being applied to
values A1, A2, ..., form the list,

( A1 A2 ... )

Now examine this list and the bound variable list, bv, in parallel.
• If bv is neither a pair or an identifier, discard the value
  list and terminate.
• If bv is an identifier, bind the value list to that identi-
  fier.
• If bv is a pair and the value list is not a pair, signal an
  error.
• If both bv and the value list are pairs, repeat the process
  on their respective CARs and CDRs.
(Note that NIL is not an identifier.)

Let me present a few examples.

( (LAMBDA U U) )
( (LAMBDA (U) U) )
( (LAMBDA (U V) (LIST U V)) )
( (LAMBDA (U V . W) (LIST U V W)) )
( (LAMBDA (U V W) (LIST U V W)) )
( (LAMBDA (U V W . X) (LIST U V W X)) )
( (LAMBDA (U V W X) (LIST U V W X)) )
( (LAMBDA (U V . V) (LIST U V)) )
( (LAMBDA (U V) (LIST U V)) )

Note that because of the "constant" rule, excess arguments are
simply discarded, not considered an error, while missing argu-
ments are treated as an error.

Once the environment corresponding to the bindings has been
created, the body of the LAMBDA expression is evaluated as an
implied PROGN.

Note that the environment created allows access to the vari-
ables of the in the environment of the expression. Thus, like
the funars, free variables will resolve to immediate bindings.

A LAMBDA expression which is found by evaluating an operator,
(an identifier, say), does not have access to the local environ-
ment. In order that a variable be accessible from a called
function it must be declared to be FLUID. This corresponds to
SPECIAL in many other LISP systems. The difference is that the
declaration of SPECIAL is made outside of any specific LAMBDA
expression, while the declaration of FLUID applies to one specific instance of a variable, in one specific LAMBDA expression.

In order to declare a variable FLUID, the position usually taken by the variable, say X, in the bound variable list is filled by the list (FLUID X). Suppose that:

\[
\begin{align*}
\text{FOO} &= \text{(LAMBDA} \ (X) \ (\text{BAZ})) \\
\text{BAR} &= \text{(LAMBDA} \ ((\text{FLUID} \ X)) \ (\text{BAZ})) \\
\text{BAZ} &= \text{(LAMBDA} \ () \ X)
\end{align*}
\]

then the following sequence would ensue.

\[
\begin{align*}
\text{(SETQ} \ X \ 10) \\
\text{Value} &= 10 \\
\text{(FOO} \ 1) \\
\text{Value} &= 10 \\
\text{(BAR} \ 1) \\
\text{Value} &= 1
\end{align*}
\]

In the first case, (FOO 1), the free variable, X, in BAZ does not see the binding of X in FOO, while in the second case, (BAR 1), it does see the FLUID binding in BAR. This visibility of the FLUID binding is not just limited to the immediately called function, but is also effective at deeper levels of calling.

A LAMBDA expression treated as an expression, and not as an operator, evaluates to a funarg. The expression portion of the funarg is simply the original LAMBDA expression, while the state-descriptor portion captures the environment in which the LAMBDA expression was evaluated.

When the resulting funarg is applied, free variables in the LAMBDA expression will be resolved, lexically, in the environment of the state-descriptor.

Suppose the value of the identifier FOO is

\[
\text{(LAMBDA} \ (X) \ (\text{LAMBDA} \ (Y) \ (\text{PLUS} \ X \ Y)))
\]

and the expression (FOO 7) is evaluated.

The value of this expression will be

\[
\text{X.FUNARG.}((\text{LAMBDA} \ (Y) \ (\text{PLUS} \ X \ Y)) \ . \ \%SD.xxxxxxxxxx)
\]

where the sd part of the funarg captures the binding of X (from the original LAMBDA expression) to 7. The resulting funarg will now add 7 to any argument that it is applied to.

**MLAMBDA — special form**

\[
\text{(MLAMBDA bv-list \ [exp ...])}
\]

Like the LAMBDA, the MLAMBDA expression behaves differently when used in operator or operand position.

An MLAMBDA expression, used as an operator, specifies a macro application. This differs from ordinary application (such as of a LAMBDA expression) in three ways.

1. No evaluation of operands occurs before the macro application.
2. The entire expression, including the (unevaluated) operator, becomes the operand of the macro operator.
3. The value returned by the macro operator is itself evaluated, as if it had occurred in the place of the original expression.
The expression (which becomes the operand) is treated as if it were the conceptual operand list described above, under LAMBDA. Thus to bind the entire expression to a variable, one would write

```
((MLAMBDA X
    (PRETTYPRINT (LIST "XXX X XXXX))
    (CONS "CAR (CDR X))
  "(A . B))
"XXX)
Value = A
```

It is as if an outer pair of parentheses had been supplied. One can use a structured bound variable list, as well.

```
((MLAMBDA (() . X)
    (PRETTYPRINT (LIST "XXX X XXXX))
    (CONS "CAR X))
  "(A . B))
"XXX ("(A . B)) XXX)
Value = A
```

In these example we see the side effects of the macro application, and the value of the resulting expression, but not the value of the macro itself. In both cases that is

```
(CAR "(A . B))
```

but that expression is reevaluated before any value is returned.

As in the LAMBDA expression, an MLAMBDA which is written explicitly as an operator has access to the lexical variables of its environment.

This is true of an MLAMBDA expression which is the result of a macro application. Thus, one macro application may create a new expression which contains a macro operator, resulting in a second macro application.

An MLAMBDA expression which is the immediate value of an operand does not have such lexical access.

If an MLAMBDA expression is found as a result of repeated evaluations of an operator, an error break is taken. The rational for this action is that elements of the expression, putative operands, will have been evaluated, making it impossible to recover the original form, the correct operand of the MLAMBDA expression.

This rule applies to bpi's as well.

If an operand which the compiler assumed would have an applicable value at execution time has, in fact, a macro applicable operand, the same error break is taken.

CLOSEDFN — special form

(CLOSEDFN item)

This operator is exactly equivalent to QUOTE during interpretation. It acts as a signal to the compiler to compile its operand. This allows a QUOTEd bpi to be constructed. In compiled code, the value of

```
(CLOSEDFN (LAMBDA (X) (PLUS X 7)))
```

is a bpi, which adds 7 to its argument. This differs from

```
(QUOTE (LAMBDA (X) (PLUS X 7)))
```

54 YKTLISP Program Description and Operations Manual
which is the specified list structure, and

\[(\text{LAMBDA} \ (x) \ (\text{PLUS} \ x \ 7))\]

which results in a compiled function, a bpi, but one which is enclosed in a funarg.

**FUNARG — special form**

\[(\text{FUNARG} \ \text{exp} \ \text{sd})\]

Something of a fossil. This operator creates a funarg (primitive object) comprised of its two arguments, unevaluated.

In an earlier version of LISP/370 the funarg object was not a primitive, but was rather a list, \((\text{FUNARG} \ \text{expression} \ \text{sd})\). In order to allow older code, which constructed such funarg, to run in the later system, it was necessary to define the operator FUNARG in this manner.

**FUNCTION — special form**

\[(\text{FUNCTION} \ \text{exp})\]

This is also an anachronism. It is exactly equivalent to \((\text{CLOSURE} \ "\text{exp} \ \text{STATE})\). It is included for compatibility with older programs, written when CLOSURE did not exist, and when the mapping operators were functions rather than macros.

**MULTIPLE LEVEL RETURNS**

Each application of an ordinary operator (LAMBDA expression or bpi) results in the creation of a frame in the stack. Normally, execution remains within a frame until the end of the expression sequence of a LAMBDA body, or an explicit RETURN, causes execution to revert to the immediately preceding frame.

There exists a set of operators which mark certain stack frames as catch points. A catch point is a frame which can receive control directly from a frame which is not its immediate successor, via the operator THROW. The frame passing control to a catch point must be a successor of the catch point, but may be many level below it.

Certain catch points will intercept only specific THROW operations, others will intercept a wide class, or even all THROW operations.

Each THROW has two operands, the first (referred to as the tag) specifying the targeted catch point, the second providing an arbitrary value which is available when the (referred to as the value) THROW terminates. When a catch point receives control it has access to both of these values. It may then execute arbitrary code, continue execution, or propagate the THROW, with the same or modified tag.

Each catch point corresponds to a stack frame, with variable bindings. In particular each such frame has a lexical variable \text{CATCH.MESS} which is bound to a "message", a distinctive value identifying the catch point. These values are found and interpreted by the \text{CATCH} operator. See page 135 and Figure 14 on page 56.

**CATCH — macro**

\[(\text{CATCH} \ id1 \ \text{exp} \ [\text{id2} \ \text{item} \ \ldots])\]

Where neither \text{id1} nor \text{id2} are evacuated, but are used as written.

\text{CATCH} establishes a catch point and then evaluates \text{exp}. If no THROW occurs during the evaluation the value of the \text{CATCH} expression is the value of \text{exp}. If a THROW to a tag \text{EQ} to \text{id1} occurs, the value is the value THROWN.
ERRSET
(0. [user-item ...]) Return THROWen value.

ERRCATCH
(2. [user-item ...]) Return THROWen value.
(3. [user-item ...]) Return THROWen value, sets FLAGVAR.

CATCH
(4. [user-item ...]) Return THROWen value.
(5. [user-item ...]) Return THROWen value, sets FLAGVAR.

NAMEDERRCATCH
(6. [user-item ...]) Return THROWen value.

SUPERMAN
901 Call to SUPV
Tries again, with existing streams.
EVAL of user provided expression, INITSUPV
Same action as 901.

902

SUPV
903 Read input
Returns to read with existing streams.
904 Echo-print
Same action as 903.
905 EVALFUN of input
Same action as 903,
but ,VAL updated with caught value.
Value-print
Same action as 903.
906 Print of message when UNWIND is caught
Same action as 903.

907

EXF
908 SUPV call
EXFTEMP LISPLIB not renamed (if it
exists), files shut.

ERRORLOOP
909 Error message print
Enters the read loop.
Read input
Returns to input read.
910 911 EVAL input
Same as 910.
Value-print
Same as 910.
912 913 Print of message when UNWIND is caught
Same as 910.

914

DISPATCHER
914 Dispatch loop
Stops interrupt scheduler scan.

\* marks catch points which intercept numeric tags and count them down.

Figure 14. CATCH, MESS values, actions and interpretation.

If id2 is present, it is interpreted as a variable, and is used...
to indicate the mode of return. If the evaluation of \texttt{EXP} completes normally, \texttt{id2} is assigned a value of \texttt{NIL}. If a \texttt{THROW} to \texttt{id1} prematurely terminated the evaluation, \texttt{id2} is assigned a value of \texttt{id1}. (Of course, if a \texttt{THROW} to some other tag occurs, control never returns to this \texttt{CATCH}.)

The evaluation of \texttt{EXP} is performed in such a way as to contain the scope of \texttt{EXIT} and \texttt{RETURN} expressions within it. Either will provide a value for \texttt{EXP}, but neither will cause control to leave the \texttt{CATCH} without its setting the value of \texttt{id2}, if present.

The items, if present, are made part of the value of \texttt{CATCH}, \texttt{MESS}. If \texttt{id2} is \texttt{NIL} items may follow it, but it is treated as not present.

**THROW — function**

\[\text{(THROW \{id | s-int \& item\})}\]

The \texttt{THROW} operator terminates the current evaluation and searches backwards in the stack for a catch point. At each catch point found the value of the tag of the is examined to determine whether the destination has been reached. If so, the \texttt{THROW} is stopped and execution resumes at that catch point. If not, the \texttt{THROW} is continued, possibly with a modified tag, or after arbitrary clean-up code has been executed, see \texttt{THROW-PROTECT}, below.

At the catch point which stops the \texttt{THROW} the value of \texttt{item} is available. The value of the tag is sometimes available.

**UNWIND — function**

\[\text{(UNWIND \{s-int \{item\}\})}\]

The \texttt{UNWIND} operator executes a \texttt{THROW} with a numeric tag. \texttt{s-int} defaults to 1 (one), while \texttt{item} defaults to \texttt{NIL}.

\texttt{UNWIND} is normally used to escape from the error break loop, passing control back to an error-expecting catch point.

**THROW-PROTECT — macro**

\[\text{(THROW-PROTECT \texttt{exp1 \texttt{exp2}})}\]

This operator evaluates its operands in sequence, establishing a catch point during the evaluation of \texttt{exp1}.

If the evaluation of \texttt{exp1} completes normally, its value is reserved, and becomes the value of the \texttt{THROW-PROTECT} expression, after \texttt{exp2} has been evaluated.

If the evaluation of \texttt{exp1} results in a \texttt{THROW} which is not caught before control passes the \texttt{THROW-PROTECT}, the \texttt{THROW} is temporarily stopped, \texttt{exp2} is evaluated, and the \texttt{THROW} is continued.

The evaluations of both \texttt{exp1} and \texttt{exp2} are protected against control being lost due to \texttt{RETURN} or \texttt{EXIT} expressions. A \texttt{THROW} in \texttt{exp2} will, however, take precedence over the resumption of the suspended \texttt{THROW}.

**ERRSET — macro**

\[\text{(ERRSET \texttt{exp \{item \ldots\}})}\]

The \texttt{ERRSET} operator establishes a catch point and evaluates \texttt{exp}. If the evaluation terminates normally the value of the \texttt{ERRSET} expression is a list of one element, containing the value of \texttt{exp}.

If a \texttt{THROW} with a numeric tag occurs during the evaluation, \texttt{ERRSET} examines the tag. If the tag is 0 (zero), the \texttt{THROW} is stopped and the value of the \texttt{ERRSET} expression is the value \texttt{THROWN}. If the tag is not 0 (zero) the \texttt{THROW} is continued, with the tag decremented by 1 (one).
Thus, the first operand of the THROW (or the UNWIND) controls the number of ERRSET (and ERRCATCH and NAMEDERRSET) expression to be skipped.

The items, if present, are make part of the value of CATCH.MESS.

**NAMEDERRSET — macro**

(NAMEDERRSET id exp [item ...])

NAMEDERRSET behaves like ERRSET, but allows a tag and user messages to be specified.

This allows an UNWIND directly to a specific NAMEDERRSET, even when there may be an unknown number of ERRSETs or ERRCATCHs intervening.

The items, if present, are make part of the value of CATCH.MESS.

**ERRCATCH — macro**

(ERRCATCH exp [id [item ...]])

ERRCATCH behaves like ERRSET with two exceptions. First, the value of the ERRCATCH expression is either the value of exp (not in a list), or the value of the intercepted THROW. Second, id, if present, is set to NIL if the evaluation of EXP terminates normally, and to 0 (zero) if a THROW is intercepted.

Unlike ERRSET, which can be fooled by THROWing a list as value, ERRCATCH allows the return of arbitrary values, while still detecting abnormal returns.

The items, if present, are make part of the value of CATCH.MESS. If id is NIL items may follow it, but it is treated as not present.
ENVIRONMENT OF EVALUATION

As was pointed out in the introduction to the previous section, the two parts of the KTDLISP environment are tightly intertwined. LAMBDA expressions (and corresponding bp's) always affect both parts. These aspects are described in the previous section.

This section will describe those operators which primarily affect the environment of evaluation alone.

EVALUATION

EVAL — built in function

(EVAL exp)

The EVAL operator causes the value of its operand, exp, to be evaluated as if it had occurred in the place of the EVAL expression.

(SETQ X "A")
Value = A
((LAMBDA (A B) (EVAL B)) 1 X)
Value = 1

The sequence of evaluations is:
1. X evaluates to A
2. The values of A and B become 1 and A, respectively
3. B evaluates to A
4. EVAL applied to A evaluates to 1
5. The LAMBDA expression returns 1

Note well, that even though the identifier A was not lexically present in the LAMBDA expression, its lexical value was found.

EVAIFUN — function

(EVAIFUN exp)

The EVAIFUN operator causes the value of its operand, exp, to be evaluated as if it had occurred in a function of no arguments called from the place of the EVAIFUN expression.

(SETQ X "A")
Value = A
(SETQ A 10)
Value = 10
((LAMBDA (A B) (EVAIFUN B)) 1 X)
Value = 10

The sequence of evaluations is:
1. X evaluates to A
2. The values of A and B become 1 and A respectively
3. B evaluates to A
4. EVAIFUN applied to A evaluates to 10
5. The LAMBDA expression returns 10

Here, unlike the previous case, the lexical variable, A, in the LAMBDA expression is not found by the evaluation, and the global value is seen.

EVAL — built in function

(EVAL exp sd)

EVAL causes the value of exp to be evaluated in the environment captured by sd. This evaluation, like that of EVAL, has access to the lexical binding in sd.
(SETQ X (LAMBDA (Y) (STATE)) 10)
Value = x
(SETQ Y 100)
Value = 100
(EVAL "x")
Value = 10
(EVAL "(PLUS Y Y) X")
Value = 20

EVAL-ID — built in function

(EVAL-ID id)
The value of this operator is exactly the same as that of EVALFUN when applied to the same argument. It differs in that its operand must have an identifier as value, (there is no restriction on the value of the identifier, just on the value of the operand in the EVAL-ID expression), and in its efficiency. Where EVALFUN must be able to evaluate any expression, EVAL-ID, being restricted to identifiers, can use a much faster mechanism.

EVAL-LEX-ID — built in function

(EVAL-LEX-ID id)
This operator is the (special case) identifier evaluator corresponding to EVAL. Unlike EVAL-ID it can access those lexical bindings which are visible from the context of application.

(SETQ X (SETQ Y 10))
Value = 10
((LAMBDA (X (FLUID Y)) (EVAL-ID "X")(Y)) 100 100)
Value = 10
((LAMBDA (X (FLUID Y)) (EVAL-LEX-ID "X")(Y)) 100 100)
Value = 100
((LAMBDA (X (FLUID Y)) (EVAL-ID "Y")(Y)) 100 100)
Value = 100
((LAMBDA (X (FLUID Y)) (EVAL-LEX-ID "Y")(Y)) 100 100)
Value = 100

EVAL-GLOBAL-ID — function

(EVAL-GLOBAL-ID id)
This operator returns the value of its argument as bound in the current global environment. No values bound on the stack, either FLUID or LEXical, are seen. If no binding is found in the current global environment the value of the expression is id.

CEVAL-ID — function

(CEVAL-ID id)
This operator searches the stack for a FLUID binding of the value of its operand, following the control chain. If no binding is found in the stack, no global environment search is made and the value of the expression is id.

As was explained (see page 41) the stack may split, with the control chain indicating the hierarchy of applications, while the environment chain indicates the search order for evaluation. CEVAL-ID breaks the evaluation search rule, and follows the application hierarchy at the splits.

The primary purpose of this abnormal evaluation is to search the control chain for values which control exception handling. The error break mechanism is controlled by the value of a free (hence global or FLUIDly bound) variable, PROGRAM-EVENTS.

It is felt that the error break specifications found in the control chain are more meaningful than those found in
the environment chain in the cases when they differ. Thus, a use of EVAL will not temporarily switch exception handling to the status which existed when the 5d operand was created, but will retain the current status.

CEVAL-LEX-ID — function

(CEVAL-LEX-ID id)

This operator differs from CEVAL-ID only in having access to those lexical bindings visible from the application context.

ASSIGNMENT

SETQ — special form

(SETQ id exp)

The assignment operator. The value of exp is assigned to id. While exp is evaluated, id is not, and must be an identifier, otherwise an error break is taken.

COUNT
Value = COUNT
(SETQ COUNT "100"
Value = 100
COUNT
Value = 100

RESETQ — macro

(RESETQ id [item])

Assigns the value of item to id, as in SETQ. The value of this expression is the value which id had before the assignment.

item defaults to NIL.

(SETQ X "(1 2 3)"
Value = (1 2 3)
(RESETQ X (CDR X))
Value = (1 2 3)
X
Value = (2 3)

SET — built in function

(SET id exp)

The other assignment operator. The value of exp is assigned to id. Unlike SETQ, both arguments are evaluated. The value of id must be an identifier, otherwise an error break is taken.

COUNT
Value = COUNT
(SETQ X "COUNT"
Value = COUNT
(SET X "100"
Value = 100
COUNT
Value = 100
X
Value = COUNT

SET-ID — function

(SET-ID id item)

This operator is equivalent to SET, except that it affects only FLUID or global bindings.

SET-LEX-ID — function

Environment of evaluation
(SET-LEX-ID id item)

An anomaly. Differs from SET only in being a function rather than a built-in operator. Oddly enough this makes SET-LEX-ID faster, as the current implementation of SET involves starting up the entire interpreter.

SET-GLOBAL-ID — function

(SET-GLOBAL-ID id item)

This operator performs an assignment in the current global environment. It does not affect any stack bindings. If no binding for id is found one is added to the head of the current global environment.

CSET-ID — function

(CSET-ID id item)

This operator is the assignment operator corresponding to CEVAL-ID. It searches the control chain for a FLUID binding of id, and if it finds one updates it with item. If no binding is found in the control chain no updating is performed and the value of the CSET-ID expression is id. No search of a global environment is made.

CSET-LEX-ID — function

(CSET-LEX-ID id item)

The assignment operator corresponding to CEVAL-LEX-ID. Differs from CSET-ID in that it has access to those LEXical bindings which are visible in the application context.
ITERATION OVER LISTS AND VECTORS

Mapping operators are the commonest iterative constructs in LISP. All LISP systems have several, usually with conflicting definitions. In YKLISP we support three of the LISP1.5 mapping operators, six of the MACLISP mapping operators (with slightly modified names) and several additional ones.

All of the YKLISP mapping operators are macros, which construct PROGs, with their functional operand explicitly placed in operator position. This obviates the need to construct functions, with their overheads of state descriptors, and allows many macros to be used as functional operands.

In general, LISP mapping operators apply a functional operand to successive portions of one or more lists, often constructing a value from the results of these applications. In addition, YKLISP provides operators which iterate over vectors, and others which will terminate their iteration when some criterion is met.

All of the mapping operators terminate when their shortest list or vector is exhausted.

This allows their use with "infinite" lists, e.g. the value of the CONS operator, as long as one of their arguments is non-cyclic.

The chief difference between the mapping operators as defined in LISP1.5 and as defined in MACLISP is the order of arguments. A LISP1.5 mapping expression is of the form

(mapping-operator list app-0b)

while a MACLISP mapping expression is of the form

(mapping-operator app-0b list ...)

It is evident that the former allows only function of one argument, while the latter allows multi-argument functions.

As an earlier version of LISP (LISP360) in use at Yorktown Heights used the LISP1.5 operators we have retained their syntax. The MACLISP operators are formed by prefixing their usual names by an M, yielding MMAP, MMAPCAR, etc.

MACLISP STYLE OPERATORS

The MACLISP style operators, the MMAP... operators, can be split along two axes.

- Those that apply their app-0b to the successive elements of their list operands.
  - MMAPC, MMAPCAR, MMAPCAN
- Those that apply their app-0b to their list operands, and the successive CD...Rs of those operands.
  - MMAP, MMAPLIST, MMAPCON

and

- Those that return their first list operand as their value.
  - MMAPC, MMAP
- Those that construct a list of the values returned by successive applications of their app-0b.
  - MMAPCAR, MMAPLIST
- Those that use NCONE to splice together the values returned by successive applications of their app-0b.
  - MMAPCONE, MMAPCON

Iteration over lists and vectors  63
MKAPC — macro

(MMKAPC app-ob list ...)

MMAPC applies app-ob to the 1st, 2nd, etc. elements of the lists. The value of the expression is the first list.

(MMKAPC (LAMBDA (X Y) (PRINT (CONS X Y))) "(1 2 3 4) "(9 8 7))
   (1 . 9)
   (2 . 8)
   (3 . 7)
   Value = (1 2 3 4)

MMAP — macro

(MMAP app-ob list ...)

MMAP applies app-ob to the lists, then to the CDRs, CADDRs, etc. of the lists. The value of the expression is the first list.

(MMAP (LAMBDA (X) (PRINT X)) "(1 2 3 4))
(1 2 3 4)
(2 3 4)
(3 4)
(4)
Value = (1 2 3 4)

Note that the expression

(MMAP PRINT "(1 2 3 4))

would have the same result.

These two operators are used only for their side effects, in this case, printing.

MMAPCAR — macro

(MMAPCAR app-ob list ...)

MMAPCAR applies app-ob to the 1st, 2nd, etc. elements of the lists. The value of the expression is a list of the values of these applications.

(MMAPCAR (LAMBDA (X Y) (CONS X Y)) "(1 2 3 4) "(9 8 7))
Value = ((1 . 9) (2 . 8) (3 . 7))

Remember that YKLISP discards any extra operands. Thus, one can use the termination condition of the mapping operators, when the shortest list is exhausted, in many ways.

For instance:

(SETQ X "(1 2 3 4))
Value = (1 2 3 4)
(MMAPCAR (LAMBDA () ()) X)
Value = ((()) () ()())

Here we have constructed a list of NILs, equal in length to another list.

In order to facilitate such operations a group of auxiliary operators has been provided, NILFN, TRUEFN and IDENTITY. See page 69.

MMAPLIST — macro

(MMAPLIST app-ob list ...)

MMAPLIST applies app-ob to the lists, then to the CDRs, CADDRs, etc. of the lists. The value of the expression is a list of the values of these applications.
(MMAPLIST IDENTITY "(1 2 3 4))
Value = ((1 . XL1=(2 . XL2=(3 . XL3=(4)))) XL1 XL2 XL3)

Figure 15. Result of MMAPLIST.

MMAPCAN — macro

(MMAPCAN app-ob list ...)

MMAPCAN applies app-ob to the 1st, 2nd, etc. elements of the lists. The value of the expression produced by MCONCing the values of these applications together.

(MMAPCAN LIST "(1 2 3) "(6 7 8 9))
Value = (1 6 2 7 3 8)
(SETQ X (LIST "(1) "(4 5) 8 "(2)))
Value = (((1) (4 5) 8 (2))
(MMAPCAN IDENTITY X)
Value = (1 4 5 2)
X
Value = ((1 . XL1=(4 5 . XL2=(2))) XL1 & XL2)

Figure 16. Result of MMAPCAN.

With both MMAPCAN and MMAPCON side effects can be drastic. Unless you are constructing the values of the individual applications, ab novo, you should be very careful.

MMAPCON — macro

(MMAPCON app-ob list ...)

MMAPCON applies app-ob to the lists, then to the CDRs, CDDRs, etc. of the lists. The value of the expression produced by MCONCing the values of these applications together.

(MMAPCON COPY "(1 2 3 4))
Value = (1 2 3 4 2 3 4 3 4 4)

MMAPLACA — macro

(MMAPLACA app-ob list ...)

Iteration over lists and vectors  65
MMAPLACA applies \textit{app-ob} to the 1st, 2nd, etc. elements of the lists. The first list is updated, with RPLACA, with the values of the successive applications. The value of the expression is the first (updated) list.

\begin{verbatim}
(SETO X "(1 2 3 4))
Value = (1 2 3 4)
(MMAPLACA ADD1 X)
Value = (2 3 4 5)
X
Value = (2 3 4 5)
(MMAPLACA CONS X (LOTSOF a b))
Value = ((2 . a) (3 . b) (4 . a) (5 . b))
X
Value = ((2 . a) (3 . b) (4 . a) (5 . b))
\end{verbatim}

\textbf{SCANOR — macro}

(SCANOR app-ob list ...)

SCANOR applies \textit{app-ob} to the 1st, 2nd, etc. elements of the lists. The iteration terminates at the first such application which results in a non-NIL value. If all applications result in NIL values, the value of the expression is NIL, otherwise it is the non-NIL, terminating value.

\begin{verbatim}
(SCANOR (LAMBDA (X) (GREATERP X 10)) "(5 3 13 8 23))
Value = 13
(SCANOR (LAMBDA (X) (GREATERP X 100)) "(5 3 13 8 23))
Value = ()
\end{verbatim}

ASSQ (see page 86) could be defined as

\begin{verbatim}
(LAMBDA (I L)
  (SCANOR
    (LAMBDA (X)
      (COND
        ((AND (PAIRP X) (EQ (CAR X) I))
         X))
    L))
\end{verbatim}

\textbf{SCANAND — macro}

(SCANAND app-ob list ...)

SCANAND applies \textit{app-ob} to the 1st, 2nd, etc. elements of the lists. The iteration terminates at the first such application which results in a NIL value. If all applications result in non-NIL values, the value of the expression is the value of the last application, otherwise it is NIL.

\begin{verbatim}
(SCANAND NUMBERP "(1 2 3 4 5)"
Value = 5
(SCANAND NUMBERP "(1 2 a 3 b)"
Value = ()
\end{verbatim}

\textbf{MACLISP STYLE OPERATORS FOR VECTORS}

The following operators are designed primarily for iteration of vector operands. Since they use \textit{ELT} to access their operands, they will work correctly on lists, as well as on strings. Since they require an upper limit on their index value, the compute the \textit{MIN} of SIZE of all of their operands. This restricts their use to non-cyclic lists. In addition, since \textit{SIZE} must traverse its operand when it is a list, such operands will be traversed twice.

The upshot of all this is, only use these operators when one or more of the arguments are vectors, and never use them with circular lists.

In addition, note that these operators have names without the M pre-
fix.

66 YKTLISP Program Description and Operations Manual
MAPE — macro

(MAPE app-ob vec ...)

MAPE applies app-ob to the 1st, 2nd, etc. elements of the vecs. The value of the expression is the length of the shortest vec, i.e., the number of iterations.

(MAPE PRINT 'ABC')
A
B
C
Value = 'ABC'

MAPELT — macro

(MAPELT app-ob vec ...)

MAPELT applies app-ob to the 1st, 2nd, etc. elements of the vecs. The value of the expression is a reference vector containing the values of the successive applications.

(MMMAPELT
(LAMBDA (X Y) (STRCONC X Y))
'ABCD'
'1' '23' '456' '7890' '12345')
Value = '<A1' 'B23' 'C456' 'D7890'>

MAPSETE — macro

(MAPSETE app-ob vec ...)

MAPSETE applies app-ob to the 1st, 2nd, etc. elements of the vecs. The first vec is updated by SETELT with the successive value of the applications. The value of the expression is the first (updated) vec.

(SETQ X <1 2 3 4>)
Value = <1 2 3 4>
(MAPSETE PLUS X "(10 20 30 40))
Value = <11 22 33 44>
X
Value = <11 22 33 44>

These are the vector counterparts of SCANOR and SCANAND. The previous discussion applies.

VSCANOR — macro

(VSCANOR app-ob vec ...)

VSCANOR applies app-ob to the 1st, 2nd, etc. elements of the vecs. The iteration terminates at the first such application which results in a non-NIL value. If all applications result in NIL values, the value of the expression is NIL, otherwise it is the non-NIL, terminating, value.

(SETQ X 'This, is a test.')
Value = 'This, is a test.'
(VSCANOR (LAMBDA (X) (MEMQ X "( . , ; : ))) X)
Value =

VSCANAND — macro

(VSCANAND app-ob vec ...)

VSCANAND applies app-ob to the 1st, 2nd, etc. elements of the vecs. The iteration terminates at the first such application which results in a NIL value. If all applications result in non-NIL values, the value of the expression is the value of the last application, otherwise it is NIL.
(VSCANAND LESSP <3 6 4 8> <5 7 8 10>)
Value = #T
(VSCANAND LESSP <3 6 4 8> <5 7 2 9>)
Value = NIL

LISP 1.5 STYLE MAPPING OPERATORS

Since the semantics of these operators is a subset of the semantics
of the MACLISP style operators, they are simply defined as macros
which change the order of the operands and replace the operator by
the corresponding M prefixed operator.

MAP — macro

(MAP list app-ob)
(MMAP app-ob list)

MAPCAR — macro

(MAPCAR list app-ob)
(MMAPCAR app-ob list)

MAPLIST — macro

(MAPLIST list app-ob)
(MMAPLIST app-ob list)

MISCELLANEOUS

MAPOBLIST — function

(MAPOBLIST app-ob)

MAPOBLIST applies app-ob to each identifier in the object
array, that is to each non-GENSYM, INTERNed identifier in the
system. The value of the expression is NIL, so any results must
be obtained via side effects.

WRAP — function

(WRAP list item)

This operator iterates over list, creating a new list whose ele-
ments are determined by the value of item.

For list of the form

(i1 i2 ...)

if item is NIL, the value will be list,
if item is not a pair, the value will be

((item i1) (item i2) ...)

if item is a list, its elements are match with the elements of
list, with the value containing elements from list where the
remaining elements of item is NIL, and lists of the elements
from item and list otherwise.

If item is a list, and if list contains more elements than item,
the final CDR of item is matched against the remaining elements
of list.

68 YKTLISP Program Description and Operations Manual
(WRAP "(A B C D E) "QUOTE)
Value = ("A "B "C "D "E"
(WRAP "(A B C D E) "FLUID)
Value = ((FLUID A) (FLUID B) (FLUID C) (FLUID D) (FLUID E))
(WRAP "(A B C D E) "FLUID () () FLUID))
Value = ((FLUID A) B C (FLUID D) (FLUID E))
(WRAP "(A B C D E) "(X Y Z))
Value = ((X A) (Y B) (Z C) D E)

AUXILIARY OPERATORS

These are operators which correspond to often used app-ob operands. They
have been defined both for a clearer style, since their names are mar-
ginally more understandable than the corresponding LAMBDA expressions,
and in order to obtain more efficient code, via macro definitions avail-
able at compilation.

IDENTITY — function + compiler macro

IDENTITY item)
The value of IDENTITY is item. This operator is exactly equiv-
alent to

(LAMBDA (X) X)

TRUEFN — function + compiler macro

(TRUEFN)
The value of TRUEFN is #T#. This operator is exactly equivalent
to

(LAMBDA () #T#)

Remember, for this and the following, that the value of
extra operands are ignored.

NILFN — function + compiler macro

(NILFN)
The value of NILFN is NIL. This operator is exactly equivalent
to

(LAMBDA () ())
DATA TYPES, TYPE TESTING AND OTHER PREDICATES

YKLISP allows the creation and manipulation of data objects of a number of different types; see "YKLISP data types" on page 23.

There exist a collection of operators which are used to determine, at execution time, the type of any particular data object. These operators are usually referred to as predicates, and the majority have names ending in P. Each of the primitive types has a corresponding predicate. In addition some unions of primitive types have predicates, as do some arbitrary subsets.

These are by no means the only predicates. There are many two argument operators which act as predicates (EQUAL and GREATERP for example), as well as operators which report on various aspects of the state of the system (IOSTATE and BOUNDP). This section describes only those which deal with the type of the object directly.

It must be remembered that the primary use of predicates is in COND expressions (or such forms as OR and AND, which are variations on COND), and that in COND expressions truth and falsity are represented by non-NIL and NIL. Most predicates will return their value for "true", if it is appropriate. (Exceptions to this rule are NULL, ATOM and LISTP, all of which are true when their argument is NIL.)

GENERAL

NONSTOREDP — function + compiler macro

(NONSTOREDP item)

Returns item if item is not a stored object. By "stored" we really mean, possibly heap resident, and hence, may be moved by the garbage collector. This is indicated, in a somewhat arbitrary way, that the type code of item has a zero in its high-order bit. Thus, small integers, generated symbols, binary programs are examples of non-stored objects.

NIL AND TRUTH VALUE

NULL — built-in function

(NULL item)

This function has the value *T* if item is NIL; otherwise, it returns the value NIL.

NOT — function + compiler macro

(NOT item)

NOT is exactly equivalent to NULL. Its reason for existence, other than historical, is to allow a more meaningful operator when the predicate in a COND is to be inverted.

In some sense it doesn't belong in this section, but its equivalence to NULL made it seem the best place for it.

PAIRS AND LISTS

ATOM — built-in function

(ATOM item)
This function returns the value NIL if item is a pair; otherwise, value is T.

It is unfortunate that the word ATOM is wasted as the not-pair predicate, but to change this tradition would lead to considerable confusion and problems of compatibility.

LISTP — built in function

(LISTP item)

This function returns the value T if item is a list; otherwise, it returns the value NIL. The value of LISTP applied to NIL (the empty list) is T.

PAIRP — built in function

(PAIRP item)

This function returns item if item is a pair; otherwise, it returns the value NIL. It is distinguished from LISTP by the fact that (LISTP NIL) = T, whereas (PAIRP NIL) = NIL.

VECTORS, STRINGS AND BPISS

REFVECP — built in function

(REFVECP item)

Returns item if it is a reference vector, else returns NIL.

WORDVECP — built in function

(WORDVECP item)

Returns item if it is a vector of 32 bit integers, else returns NIL.

REALVECP — built in function

(REALVECP item)

Returns item if it is a vector of 64 bit floating point numbers, else returns NIL.

STRINGP — built in function

(STRINGP item)

This function returns the value item if item is a character string (that is, a vector of characters); otherwise, it returns the value NIL.

BITSTRINGP — built in function

(BITSTRINGP item)

This function returns the value item if item is a bit string (i.e., a vector of bits); otherwise, it returns the value NIL.

SUBRP — built in function

(SUBRP item)

Returns item if item is a compiled function, otherwise returns NIL.

MSUBRP — built in function

(MSUBRP item)
Returns item it item is a compiled macro (MLAMBDA ...) expression, otherwise returns NIL.

**BPIP — function + compiler macro**

(BPIP item)

Returns item if item is a bpi, that is either a subr or an msubr, otherwise returns NIL.

Here you see the terminology shifting before your very eyes.

**VECP — built in function**

(VECP item)

This function returns the value item if item is any variety of vector; otherwise, it returns the value NIL.

See Figure 21 on page 52. Besides the vector objects described in that figure, VECP accepts bpis.

**IDENTIFIERS**

**IDENTP — built in function**

(IDENTP item)

This function returns the value item if item is an identifier; otherwise, it returns the value NIL.

IDENTP will return a non-NIL value for frs, mrss, ordinary identifiers and gensyms. NIL is not an identifier, and (IDENTP "NIL") = NIL.

**FRP — built in function**

(FRP item)

Returns item if it is an fr (a built in operator); otherwise returns NIL.

The term "fr" derives from the phrase Functional operator.

**MRP — built in function**

(MRP item)

Returns item if it is an mr (a special form); otherwise returns NIL.

The term "mr" derives from the phrase Macro operator.

**GENSYMP — built in function**

(GENSYMP item)

Returns item if it is a gensym; otherwise returns NIL.

**CHARP — function + compiler macro**

(CHARP item)

This operator tests whether item is a character object; one of the 256 identifiers which span the total range of possible single character print names. The value of CHARP is NIL if item is not one of these objects, otherwise the value is item.

**DIGITP — function + compiler macro**

Data types, type testing and other predicates  73
DIGITP item

Returns item if it is one of the character objects (identifiers) 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9; otherwise, returns NIL. (Note, the actual print representation of these objects is [0 | 1 etc.)

PLACE HOLDERS

PLACEP — function + compiler macro

(PLACEP item)

Returns item as its value if item is a read-place-holder, else returns NIL.

A read-place-holder is the value of (READ x) when x is an empty stream, e.g. a stream at end-of-file. This is the only condition under which READ will return a read-place-holder as its value.

Note that there is no print representation for read-place-holders. They are gensym-like, in that they are created anew when needed.

NUMBERS

NUMBERP — built in function

(NUMBERP item)

This operator returns the value item if item is any type of number. If item is not a number, the value NIL is returned.

SMINTP — built in function

(SMINTP item)

This operator returns the value item if item is a small integer; otherwise, its value is NIL. Small integers are numbers in the range -2^t to 2^t-1.

LINTP — built in function

(LINTP item)

This operator returns the value item if item is a large integer; otherwise, its value is NIL. Large integers are an fixed point number outside the small integer range.

FIXP — built in function

(FIXP item)

This operator returns the value item if item is a fixed point number; otherwise, its value is NIL. The class of "fixed point numbers" is the union of small integers and large integers.

FLOATP — built in function

(FLOATP item)

This operator returns the value item if item is a floating point number; otherwise, its value is NIL.
FUNARGS

FUNARGP — built in function

(FUNARGP item)

If item is a funarg data object, returns item as its value; otherwise returns NIL.

STATE DESCRIPTORS

STATEP — built in function

(STATEP item)

If item is a state descriptor data object, returns item as its value; otherwise returns NIL.

STREAMS

STREAMP — function

(STREAMP item)

Returns item if it could be a stream, i.e. if it is a pair. Exactly equivalent to PAIRP.

FASTSTREAMP — function

(FASTSTREAMP item)

This function returns true if item is a fast stream -- that is, a pair whose CDR is a reference vector of length at least 3. Otherwise, the value of FASTSTREAMP is NIL.

Be warned that this is a completely heuristic test, and an arbitrary data object could "pass" it.

IS-CONSOLE — function

(IS-CONSOLE item)

This function returns NIL as its value if item is not a fast console stream. If item is such a stream, it is returned as the value of the function.

OTHER PREDICATES

EQ — built in function

(EQ item1 item2)

EQ tests for pointer identity between its two arguments. Its value is the identifier #T if item1 and item2 are identical pointers. This means that the pointer type codes as well as the pointer address fields are identical. If these fields are not identical, the value of EQ is #F.

EQ may be used for quick tests of equivalence. If two expressions are EQ, then they are necessarily EQUAL; however, the converse is not true. Two expressions which are not EQ may nevertheless be EQUAL.
Note that two copies of a given object will not be EQ because they are stored at different locations. Similarly, it is possible to have different representations of the same numeric value which are EQUAL, but which are not EQ.

**EQUAL — function**

`(EQUAL item1 item2)`

This is a generalized equality testing function applicable to any LISP objects, including circular structures and numeric quantities.

For numeric quantities to be EQUAL, they must represent the same value. For tests involving one or two real (floating point) numbers, a fuzz factor may be relevant. This is explained in the section of this manual discussing data types. If an integer is to be compared with a real number, the integer is converted to a real value for the comparison.

Two vectors are EQUAL if they are of the same type, the same length, and their absolute parts are identical and their pointer parts are EQUAL.

For composite arguments, EQUAL implements access-equivalent equality testing. This means that two structures are EQUAL if every part of one structure which can be reached by a composition of accessing functions is EQUAL to the corresponding part of the other structure reached through the same composition of accessing functions. Intuitively, two structures are EQUAL if they denote the same (possibly infinite) tree.

The value of EQUAL is either NIL or T.

See the discussion under LISTS in the Data Type section for an example and further commentary.

**UEQUAL — function**

`(UEQUAL item1 item2)`

This is a generalized update-equality testing function applicable to any LISP object in the same sense as EQUAL. It differs from EQUAL in that for two structures to be UEQUAL, not only must corresponding parts of the structures be EQUAL through the access functions, but there must be the same number of unique parts and if any of these parts were to be updated in one structure and the same update operation performed on the corresponding part of the other, then the structures would still be EQUAL.

In addition, numeric values are considered UEQUAL only if they are of the same type and numerically equivalent. Bit and character strings are UEQUAL only if they have the same capacity as well as the same type, length, and contents.

Intuitively, two structures are UEQUAL if and only if they denote equivalent rooted directed graphs, i.e. if they denote equal structures which also have the same acyclical and cyclical sharing structure.

The value of UEQUAL is either NIL or T.

See the discussion under LISTS in the Data Type section for an example and further commentary.

**EQUALN — function**

`(EQUALN item1 item2)`

The value of this function is the value of `(EQUAL item1 item2)` when the floating point fuzz factor is zero. The fuzz factor is temporarily made zero while EQUAL is invoked, then restored to its original value.
OPERATIONS ON PAIRS

These operators treat pairs as pairs, not imposing any interpretation, such as lists, on the contents. Notwithstanding, in many cases a comment is included to indicate the list interpretation of their actions.

CREATION

CONS — built in function

(CONS item1 item2)

CONS is the basic pair-creating function. Its value is the new pair constructed with item1 as its CAR component and item2 as its CDR component.

Of particular interest is the case where item2 is a pair. Then the value of CONS is the list formed by adding item1 to the beginning of the list item2.

ACCESSING

CAR — built in function

(CAR item)

One of the two basic selection functions defined on pairs. Its value is the CAR component of the pair item.

In its list interpretation, the value of (CAR item) is the first element of the list item.

If item is not a pair, an error results.

CDR — built in function

(CDR item)

One of the two basic selection functions defined on pairs. Its value is the CDR component of the pair item. If item is not a pair, an error results.

The value of (CDR item) is usually the list containing all but the first element of the list item; however, this is not the case when item is the terminating pair of a list. In that case, (CDR item) is the terminating atom, usually NIL.

The use of the words CAR and CDR is also one of those traces of history, and while FIRST and REST might be preferred (and may be defined by the user), a certain tenacious tradition causes the use of CAR and CDR to thrive.

CA|D...R — function + compiler macro

(CA|D)...R item)

There are twenty eight macros defined in YKTLISP which give meaning to operators of this form, where ... designates any sequence of one to four As orDs. For example, (CADDR item) is equivalent to

(CAR (CDR (CDR item)))

and so on. This is literally true for interpreting such a macro, but if the macro is expanded by the LISP compiler, it is smarter than that and achieves the complete operation with only
one call to an appropriate subroutine. In fact, this subroutine is capable of handling strings of CARs and CDRs up to 256 levels deep, and the macro will economize (CADR (CDR item)) to (CADDR item), and will even do well by (C...R (C...R item)) where ... is two, three or four letters, and ... is any number such that total depth is less than 256.

An error break is taken if any of the successive CAR and CDR operations applies to a non-pair.

QC(ALD)...R — function + compiler macro

(QC(ALD)...R item)

Each of the C...R operators (including CAR and CDR) has a corresponding QC...R operator. These operator perform the same action as the C...R operators, without first checking their argument for type. Thus, if applied to anything but pairs their action is unpredictable.

In compiled programs these operation are performed by in-line code.

IFCAR — function + compiler macro

(IFCAR item)

This is a conditional CAR operator. If item is a pair, it is equivalent to (CAR item), otherwise it has a value of NIL.

XORCAR — function + compiler macro

(XORCAR item)

This is another conditional CAR operator. If item is a pair, it is equivalent to (CAR item), otherwise the argument, item, is returned as the value.

IFCDR — function + compiler macro

(IFCDR item)

This is a conditional CDR operator. If item is a pair, it is equivalent to (CDR item), otherwise it has a value of NIL.

XORCDR — function + compiler macro

(XORCDR item)

This is another conditional CDR operator. If item is a pair, it is equivalent to (CDR item), otherwise the argument, item, is returned as the value.

UPDATING

RPLACA — built in function

(RPLACA pair item)

This is one of the two basic functions for updating pairs. Its value is the updated pair which results when the CAR component of the pair pair is replaced by item. An error is indicated if pair is not a pair.

QRPLACA — function + compiler macro

(QRPLACA pair item)

Equivalent to RPLACA, with no type checking of arguments.

RPLACD — built in function
(RPLACD pair item)

This is the other basic function which updates pairs. Its value is the updated pair which results when the CDR component of the pair pair is replaced by item. An error is indicated if pair is not a pair.

QRPLACD — function + compiler macro

(QRPLACD pair item)

Equivalent to RPLACD, with no type checking of arguments.

RPLACAD — function + compiler macro

(RPLACAD pair1 pair2)

This function is equivalent to

(RPLACA pair1 (CAR pair2))
(RPLACD pair1 (CDR pair2))

Its value is the updated pair, pair1. An error is indicated if either argument is not a pair.

QRPLACAD — function + compiler macro

(QRPLACAD pair1 pair2)

Equivalent to RPLACAD, with no type checking of arguments.

RPLNODE — function + compiler macro

(RPLNODE pair item1 item2)

This function is equivalent to

(RPLACA pair item1)
(RPLACD pair item2)

Its value is the updated pair. An error is indicated if the argument, pair, is not a pair.

QRPLNODE — function + compiler macro

(QRPLNODE pair item1 item2)

Equivalent to RPLNODE, with no type checking of arguments.
OPERATIONS ON LISTS

These operations assume the list interpretation of pairs, see "Lists" on page 26. The value of the final, non-pair, CDR of an argument is generally ignored once its non-pairness has been discovered. Similarly, an atomic (non-pair) argument is in most cases treated as equivalent to NIL.

There is a special form of list, called an association list, or a-list, which is used by some processes. This is a list of pairs, where the CAR of each pair is used as a name, labeling the CDR, which is interpreted as its value.

CCA 10) (B 14) (C 6))

is an association list with A having a value of 10, B of 14 and C of 6. Remember that it is the CDR of each pair which defines the value.

((1 one item) (2 two items) (0 no items at all))

is an a-list with 1 having a value of (one item), 2 of (two items) and 0 (no items at all). The order of name-value pairs in an a-list is not important, except as it effects the length of searches. In general, the first instance of a name to be found is used, thus existing values may be "shadowed" by CONSing new pairs onto the front of the list.

CREATION

LIST — function + compiler macro

(LIST [item ...])

Returns as value a list of n elements, the first element being the value of the expression item1, et cetera.

(LIST item1 item2 ... itemn)

is equivalent to

(CONS item1 (CONS item2 ... (CONS item2 NIL)))

LOTSOF — function

(LOTSOF item ...)

LOTSOF returns an infinite list of its arguments.

(LOTSOF 2 "A "(8 9))

Figure 17. Result of LOTSOF.

CONS — built-in function

(CONS item list)

When its second argument is a list, CONS can be considered a list creating operator. It is used to add new items to the beginning of a list. See also page 77.

Operations on lists
APPEND — function

(APPEND list1 list2)

If the argument list1 is not a pair, the value of APPEND is list2. If list1 is a list, the pairs constituting the list are copied and the final CDR of the copied list is set to list2. The value of APPEND then becomes this copied list. If list1 is circular, an error break is taken.

There is no copying of the structure below the top level, which would be accessed by descending the elements of list1. Consider (APPEND "(1 2) "(3 4)). The structure before, and after the call would be:

Before processing

\[
\text{list1} \quad \text{list2}
\]

\[
\rightarrow \quad \rightarrow
\]

\[
\rightarrow i1 \quad \rightarrow i2
\]

\[
\rightarrow \quad \rightarrow
\]

\[
\rightarrow i3 \quad \rightarrow i4
\]

After processing

\[
\text{list1} \quad \text{list2}
\]

\[
\rightarrow \quad \rightarrow
\]

\[
\rightarrow i1 \quad \rightarrow i2
\]

\[
\rightarrow \quad \rightarrow
\]

\[
\rightarrow i3 \quad \rightarrow i4
\]

(\text{Note, the pairs marked by 1 are newly created.})

Figure 18. Result of APPEND.

CONC — function

(CONC [list ...])

This is a macro which expands into an expression which uses APPEND to create a list from several lists. It will accept an arbitrary number of lists as arguments. For example,

(CONC list1 list2 list3) = (APPEND list1 (APPEND list2 list3))

Note that the last argument to CONC appears as the second argument to APPEND, so that it is not copied at the first level as are the other arguments to CONC, which appear as first arguments to APPEND.

REVERSE — function

(REVERSE list)

This operator returns as its value a new, top-level copy of the list list where the elements of this new list are in the inverse order of their occurrence in list.

(REVERSE "+(1 2 3 4))
Value = (4 3 2 1)

VEC2LIST — function
(VEC2LIST vec)

This operator constructs a new list, containing the elements of the vector, vec. Note that vector, in this case, includes strings, but excludes bpsi.

UNION — function

(UNION list1 list2)

This operator constructs a new list containing all the elements appearing in either of the lists list1 and list2. Each element appears only once in the resulting list. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. Any elements in the union found in list1 will precede those found only in list2. The order will match that of the lists of origin. Thus:

(UNION "(T H I S I S A T E S T)" "O F T H E U N I O N")

results in

"(T H I S A E O F U N)"

UNIONQ — function

(UNIONQ list1 list2)

This operator differs from UNION only in using MEMQ, and hence EQ, to test for membership in its arguments.

INTERSECTION — function

(INTERSECTION list1 list2)

Constructs a new list containing only those elements appearing (at the top level) in both list1 and list2. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. The elements in this new list are in the same order as their occurrence in list1. If either argument is not a pair, it is treated as if it were the only element in a list of length one.

INTERSECTIONQ — function

(INTERSECTIONQ list1 list2)

This operator differs from INTERSECTION only in using MEMQ, and hence EQ, to test for membership in its arguments.

SETDIFFERENCE — function

(SETDIFFERENCE list1 list2)

Constructs a new list containing only those elements appearing (at the top level) in list1 but not in list2. MEMBER, which in turn uses EQUAL, is used to detect whether an element appears in a list. The elements in this new list are in the same order as their occurrence in list1. If either argument is not a pair, it is treated as if it were the only element in a list of length one.

SETDIFFERENCEQ — function

(SETDIFFERENCEQ list1 list2)

This operator differs from SETDIFFERENCE only in using MEMQ, and hence EQ, to test for membership in its arguments.

MTON — function

(MTON s-int1 s-int2)
This operator returns a list of the integers from s-int1 to s-int2, inclusive. It does not check its arguments, and if either is not a small integer, or if s-int2 is less than s-int1 the results will be unpredictable, and possibly fatal.

ACCESSING

CA[D...JR — function + compiler macro

(CA[D...JR list)

Any operator of this form, that is zero or more CDRs followed by a single CAR, may be thought of as an accessing function for a list. CAR gives the first element, CADR the second, etc. See page 77.

QCA[D...JR — function + compiler macro

(QCA[D...JR list)

The non-checking versions of the preceding operators. See page 78.

CD...R — built in function

(CD...R list)

Any operator of this form, that is zero or more CDRs, may also be thought of as an accessing function for a list. Rather than giving element, however, these operators return the "tail" of their argument. CDR gives the list, less its first element, CDcdr, less its first and second, etc. See page 77.

QCD...R — function + compiler macro

(QCD...R list)

The non-checking versions of the preceding operators. See page 78.

ELT — function + compiler macro

(ELT list s-int)

While ELT is normally thought of as an operator on vectors, it also is defined on lists. Its value is the s-int element of list, where the first element is designated by zero (0), the second by one (1), etc.

If s-int is not within the bounds of list, an error break is taken. If list is not a vector or a list, an error is indicated.

Note: ELT applied to NIL will always produce a bounds error, as NIL is interpreted as the empty list.

LAST — function

(LAST list)

Returns as value the last element of list (i.e. the CAR of the last pair comprising list). If list is non-pair, the value of LAST is 0.

If list is circular an error break is taken.

LASTNODE — function

(LASTNODE list)
This operator returns as its value the last pair forming the list list. If list is non-pair or is circular an error break is taken.

```
(LASTNODE "(1 2 3 4))
Value = (4)
```

This value (as is true of the CD...R forms) is not a copy. Updating operations applied to it will effect the original list.

**SEARCHING**

**MEMBER — function**

```
(MEMBER item list)
```

This operator searches the list list for the object item, using EQUAL testing for identity. It item is not found, or if list is not a pair, the value of MEMBER is NIL. If item is found, the value of MEMBER is that portion of list beginning with item. list is searched on the top level only.

**UMEMBER — function**

```
(UMEMBER item list)
```

This operator is similar to MEMBER, except that it uses UEQUAL testing for identity instead of EQUAL testing.

**MEMQ — function**

```
(MEMQ item list)
```

This operator is similar to MEMBER, except that it uses EQ testing for identity instead of EQUAL testing.

**QMEMQ — function + compiler macro**

```
(QMEMQ item list)
```

When compiled, this operator produces inline code equivalent to the MEMQ operator. This operator does not check for cyclic lists, and will loop indefinitely.

**TAILP — function**

```
(TAILP item list)
```

This operator searches list for a CDR EQ to item. If such is found the value is XTM, otherwise it is NIL.

The case of item EQ to list is considered a success. Since EQ is used,

```
(TAILP "(3 4) "(1 2 3 4))
Value = NIL
```

will occur, as the two lists are separate. On the other hand,

```
(SETQ X "(3 4))
Value = (3 4)
(SETQ Y (APPEND X (1 2) X))
Value = (1 2 3 4)
(TAILP X Y)
Value = XTM
```

because of the sharing in the value of APPEND.

**ASSOC — function**

```
(ASSOC item a-list)
```

Operations on lists 85
a-list is a list of pairs, ((name1 . value1) (name2 . value2) ...
). ASSOC compares item with name1, then name2, etc., using
EQUAL to perform the comparison. Any elements of list which are
not pairs are skipped. If a-list is not a list, or if item is
not found in a-list, the value of ASSOC is NIL. Otherwise, the
value of ASSOC is the first pair (name . value) such that
(EQUAL item "name") is true. If there are elements of a-list
which are not pairs, they are skipped and the next element of
a-list is examined.

ASSOCN — function

(ASSOCN item a-list)

This operator is very similar to ASSOC, except that EQUALN is
used instead of EQUAL for comparing item with the CARs of ele-
ments in a-list.

UASSOC — function

(UASSOC item a-list)

This operator is identical to ASSOC except that UEQAL, rather
than EQUAL, is used for the comparison of item with the a-list.

ASSQ — function

(ASSQ item a-list)

This operator is similar to ASSOC, except it uses EQ rather than
EQUAL to compare item with the CARs of elements in a-list.

QASSQ — function + compiler macro

(QASSQ item list)

When compiled, this operator produces inline code equivalent to
the ASSQ operator. This operator does not check for cyclic
lists, and will loop indefinitely.

SAASSOC — function

(SAASSOC item a-list app-ob)

This operator is similar to ASSQ, but requires three arguments
and if item is not matched, it returns the result of applying
app-ob to no arguments, instead of the NIL value returned by
ASSOC.

GET — function

(GET list item)

The operator GET is usually used to access the property lists of
identifiers. The YKTLISP definition of GET also applies to
lists.

If list is not an identifier or a pair, value is NIL. If list is
a pair, it is interpreted as an a-list, and searched for an ele-
ment whose CAR is EQ to item. If found, the value of GET is the
CDR of the element, otherwise the value of GET is NIL.

Note the difference between GET and ASSQ, in that ASSQ
returns the name-value pair, whereas GET returns only the
value portion. Also, note the reversal of the order of
arguments; GET receives the list to be search, followed by
the name; while ASSQ (and the other .ASS. operators)
receive the name first, followed by the list to be
searched.

QGET — function + compiler macro

(QGET list item)

86 YKTLISP Program Description and Operations Manual
When compiled, this operator produces inline code equivalent to the GET operator. This operator does not check for cyclic lists, and will loop indefinitely.

SEARCHING AND UPDATING

ADDTOLIST — function

(ADDTOLIST list item)

If list is not a pair, the value returned is (LIST item). Otherwise this operator searches list for an instance of item, using EQ. If no instance of item is found, item is added to the tail of the list, using RPLACD. In effect,

(NCONC list (LIST item))

The value returned is the list list, whether or not item was initially present.

MAKEPROP — function

(MAKEPROP list item1 item2)

Just as GET, MAKEPROP is usually used to update the property lists of identifiers. And just as GET, the YKT/LISP definition of MAKEPROP also applies to lists. See page 115 for a description of MAKEPROP when applied to identifiers.

list is interpreted as an a-list. It is searched for a pair with CAR EQ to item1. If such a pair is found it is updated, with RPLACD, making item2 its CDR. If no such pair is found a new pair, (item1 . item2), is added to the front of list, by updating operations. is put at the beginning of the property list.

The value of list must be an identifier or list, but item1 and item2 may be any expressions.

REMPROP — function

(REMPROP list item)

See page 115 for a description of the behavior of REMPROP when applied to identifiers.

The item property of the identifier list is removed from the property list of list. The value of REMPROP is NIL if there is no item property. If the property exists, the value of REMPROP is the value associated with that property.

The effect is that of EFFECE, given the name-value pair as an operand.

UPDATING

RPLACA — built in function

(RPLACA list item)

When its first argument is interpreted as a list, RPLACA replaces the first element of that list.

(SETQ X "((1 2 3 4))"
Value = (1 2 3 4)
(RPLACA X "10"
Value = (10 2 3 4)
X
Value = (10 2 3 4)

Operations on lists 87
See page 78.

QRPLACA — function + compiler macro
(QRPLACA pair item)
Equivalent to RPLACA, with no type checking of arguments.

RPLACD — built-in function
(RPLACD pair item)
When its first argument is interpreted as a list, RPLACD replaces the tail of that list.

(SETQ X "((1 2 3 4))
Value = (1 2 3 4)
(RPLACD X "((20 30 40))
Value = (1 20 30 40)
X
Value = (1 20 30 40)

See page 78.

QRPLACD — function + compiler macro
(QRPLACD pair item)
Equivalent to RPLACD, with no type checking of arguments.

SETELT — function + compiler macro
(SETELT list s-int item)
Like ELT, SETELT is normally thought of as an operator on vectors, but it too is defined on lists. It is the inverse operator of ELT -- it updates the s-int element of list to be item. SETELT will take an error break if list is not updatable or if s-int is out of range.

(SETELT LIST 3 VALUE)
is equivalent to:

(RPLACA (CDR (CDR (CDR LIST))) VALUE)
The value of SETELT is its last argument, item, the value used in updating the specified object.

NCONC — function
(NCONC list1 list2)
If list1 is non-pair, the value of NCONC is list2. If list1 is a circular list, an error break will be taken. Otherwise, RPLACD is used to replace the final CDR of list1 with list2, and the updated list1 is returned as the value of NCONC.

Consider (NCONC "((i1 i2) "((i3 i4))". The structure before, and after the call would be:
Before processing

list1

After processing

value list1

(Note, no new pairs are created.)

Figure 19. Effect of NCONC

NREVERSE — function

(NREVERSE list)

This function shuffles the CDR components of the pairs making up list so that the CDR of the last pair in list points to the next-to-last pair, and so forth. The CDR of the first pair is made NIL, and the value of NREVERSE is the last pair of list, which is now the first pair of list containing exactly the same elements as list, but in reversed order. See also REVERSE, page 82, which constructs a new list in reversed order instead of reusing the pairs in the original list.

Consider (NREVERSE "(i1 i2 i3 i4)). The structure before, and after the call would be:

Before processing

list

After processing

(Note, no new pairs are created.)

Figure 20. Effect of NREVERSE

EFFACE — function

(EFFACE item list)
Uses EQUAL to search list for the first occurrence of item. If
item is not found, or if list is atomic, returns list as its
value. If item is found, it is removed from list by updating
via REPLACED, and the up-dated list is returned as the value of
EFFACE. Only the first occurrence of item will be removed from
list.

If list contains a single element, and that element is EQUAL to
item, then no updating is done, but the value of EFFACE is
(CDR list).

EFFACEQ — function

(EFFACEQ item list)

Equivalent to EFFACE, but uses EQ rather than EQUAL.

MISCELLANEOUS

LENGTH — function

(LENGTH list)

If list is a pair, LENGTH returns the number of elements in the
list beginning with that pair. If list is not a pair, the value
of LENGTH is zero.

If list is a circular list an error break is taken.

QLENGTH — function + compiler macro

(QLENGTH list)

When compiled, this operator produces inline code equivalent to
the LENGTH operator. This operator does not check for cyclic
lists, and will loop indefinitely.

SIZE — function

(SIZE list)

When applied to lists, SIZE is equivalent to LENGTH. It differs
when it is applied to vectors, in which case it returns the num-
ber of element in the vector, rather than zero, as LENGTH does.

QSORT — function

(QSORT list)

QSORT returns a newly CONSed list, containing the element of
list, ordered by the operator SORTEARTER. SORTEARTER is
initially defined as equivalent to GOURTARTER, page undefined,
but may be redefined at the users pleasure.

The quicksort algorithm is used.

SORTBY — function

(SORTBY app-qb list)

This operator sort its second argument, list. Where QSORT com-
pares the elements of the list, SORTBY compares the values of
(app-qb element). Thus where

(QSORT "((2 g) (1 n) (3 a))")
Value = "((1 n) (2 g) (3 a))"

we can have

(SORTBY "CADR "((2 g) (1 n) (3 a))")
Value = "((3 a) (2 g) (1 n))"
The set of data objects grouped under the title vector includes strings and bpsis. In this section strings will only be mentioned in passing and bpsis only once. Those operators which effect only strings are defined in their own section.

CREATE

GETREFV — function

(GETREFV s-int)

Returns as value a new reference vector containing s-int elements. The initial value of each element is $\_intil$. This is the basic allocating operator for reference vectors.

GETREALV — function

(GETREALV s-int)

Allocates and returns as value a real vector containing s-int elements. Each of the floating point values (elements) are initialized to zero.

GETWORDV — function

(GETWORDV s-int)

Allocates and returns as value a word vector containing s-int elements. The elements of the allocated vector are not initialized. See also GETZEROVEC.

GETZEROVEC — function

(GETZEROVEC s-int)

Like GETWORDVEC, except the elements of the word vector are initialized to zero.

VECTOR — function + compiler macro

(VECTOR [item ...])

This operator is the reference vector analogue of LIST. Its value is a reference vector containing its operands.

(VECTOR 1 2 3 4)
Value = <1 2 3 4>

LIST2REFVEC — function

(LIST2REFVEC list)

This operator constructs a new reference vector from the elements of list. If list is non-pair, the value of LIST2REFVEC is a reference vector with zero elements. If list is a circular list, the operator loops. Otherwise, the value is a reference vector of the form:

( (CAR LIST) (CADDR LIST) ... (CADR LIST) )

LIST2FLTVEC — function

(LIST2FLTVEC list)

This operator is similar to LIST2REFVEC but for the fact that the resulting vector is a vector of floating point (real) numbers. The elements of list may or may not already be real num-
STANDARD VECTOR OPERATORS

A tabular index to primitive vector operators

<table>
<thead>
<tr>
<th>Type of Vector</th>
<th>Predicate</th>
<th>Value of element</th>
<th>Specific Selection Function</th>
<th>Specific Updating Function</th>
<th>Allocating Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>REFVECP</td>
<td>Anything</td>
<td>ELT</td>
<td>SETELT</td>
<td>GETREFV</td>
</tr>
<tr>
<td>Character</td>
<td>STRINGP</td>
<td>Identifier</td>
<td>FETCHCHAR</td>
<td>STORECHAR</td>
<td>GETSTR</td>
</tr>
<tr>
<td>Bit</td>
<td>BITSTRINGP</td>
<td>NIL or -NIL</td>
<td>ELT</td>
<td>SETELT</td>
<td>GETBITSTR</td>
</tr>
<tr>
<td>Word</td>
<td>WORDVECP</td>
<td>Integer</td>
<td>ELT</td>
<td>SETELT</td>
<td>GETWORDV</td>
</tr>
<tr>
<td>Real</td>
<td>REALVECP</td>
<td>Real</td>
<td>ELT</td>
<td>SETELT</td>
<td>GETREALV</td>
</tr>
</tbody>
</table>

Analogous operators for pairs are listed below for comparison

<table>
<thead>
<tr>
<th>Pair</th>
<th>PAIRP</th>
<th>CAR</th>
<th>RPLACA</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>LISTP</td>
<td>Anything</td>
<td>ELT</td>
<td>SETELT</td>
</tr>
</tbody>
</table>

bers. If they are not real, they are floated. If any element of list is not a number, or cannot be converted into a floating point number, the FLOAT operator which is called by LIST2FLTVEC will take an error break.

LIST2IVEC — function

(LIST2IVEC list)

This operator is similar to LIST2REFVEC but for the facts that its value is an integer vector and the elements of list must be integers within the range acceptable for integer (word) vectors. This range is \(2^{21} > \text{value} > -2^{31}\).

ACCESSING

SIZE — function

(SIZE vec)

SIZE returns as its value the number of elements in its argument -- that is, one more than the maximum valid index which may be used to address an element of this vector. SIZE may also be applied to lists, in which case it performs exactly as does LENGTH. See page 90. If SIZE is given an argument other than a pair or vector, it returns the value zero.

Note that string vectors (either character or bit) may have a capacity larger than the number of elements currently in the vector.

You may note that while SIZE (and LENGTH) are lumped as "miscellaneous" operators when applied to lists, they are included among the access operators for vectors. This is because the number of element is a vector is explicitly recorded in the data structure, while the number of elements in a list can only be found by counting.

MAXINDEX — function

(MAXINDEX vec)

92 YKTLISP Program Description and Operations Manual
Returns as value the maximum allowed index for the given vector. This operator is defined as (SUB1 (SIZE Vec)), so while its principal use concerns vectors, it could be applied to other objects as well.

QREFVECMAXINDEX — function + compiler macro
(QREFVECMAXINDEX reference-vector)

This is the non-checking counterpart of MAXINDEX. If its operand is a reference-vector it returns the desired value, otherwise its value is unpredictable.

Note, that QREFVECMAXINDEX will produce the correct value when applied to a word vector.

QREFVECLENGTH — function + compiler macro
(QREFVECLENGTH reference-vector)

This is the non-checking counterpart of SIZE. If its operand is a reference-vector it returns the desired value, otherwise its value is unpredictable.

Note, that QREFVECLENGTH will produce the correct value when applied to a word vector.

LENGTHCODE — function
(LENGTHCODE vec)

LENGTHCODE returns as its value the size, in bytes, of the vector Vec. If Vec is a character or bit vector, this value is the maximum size specified, including the 3-byte current string length field.

An error occurs if Vec is not a vector.

QLENGTHCODE — function + compiler macro
(QLENGTHCODE vec)

This is the non-checking counterpart of LENGTHCODE. If Vec is a vector, it returns its length. If Vec is not a vector its result is unpredictable.

See also page undefined s_length...

ELT — function + compiler macro
(ELT vec s-int)

This is the general selection operator for vectors and lists. Its value is the s-int element of Vec, where the type of object returned by ELT is indicated in Figure 21 on page 92 for various types of Vecs.

If s-int is not within the bounds of Vec, an error is indicated. If Vec is not a vector or a list, an error is indicated.

The first element of a vector or a list is selected by using zero as the s-int value. Note: ELT applied to NIL will always produce a bounds error, as NIL is interpreted as the empty list.

If Vec is a vector of reals, ELT will allocate a new real number into which the value of the selected element is copied, and return this new real to the caller. This is necessary (although not very efficient) because pointers pointing inside a vector are not allowed (they confuse the garbage collector). Thus, if a collection of real numbers are to be assembled into a vector, it is better to have a reference vector when access to these reals is made on an individual basis using ELT. The vector of reals exists for applications where the user has implemented arithmetic processes requiring the contiguous storage of real data in order to execute efficiently.
IfVEC is a word vector, ELT may have to build a new large integer and return it as the value of ELT for certain values in the word vector. Any value within the range of a LISP small integer will be returned as a small integer, and will not require allocation of heap space. Values outside the range of small integers must be converted by ELT into large integers.

QREFELT — function + compiler macro

(QREFELT reference-vector s-int)

This is the non-checking counterpart of ELT. It is only defined when its first operand, reference-vector is a reference vector, and its second operand, s-int, is a small integer and is in the correct range. In that case its value is the same as ELTs. Otherwise its value is unpredictable, and possible fatal.

VEC2LIST — function

(VEC2LIST vec)

This operator constructs a new list, containing the elements of the vector, vec. The elements of the resulting list will correspond to the values of ELT for the specific vector from which they are drawn.

UPDATING

SETELT — function + compiler macro

(SETELT vec s-int item)

This is the inverse operator of ELT -- it updates the s-int element of vec to be item. The nature of item for various types of vec is indicated in Figure 21 on page 92.

SETELT will take an error break if vec is not updatable, if s-int is out of range, or if item is not compatible with the type of vec.

SETELT may be used to update the s-int element of a list.

The value of SETELT is item, the value used in updating the specified object.

QSETRFV — function + compiler macro

(QSETRFV reference-vector s-int item)

This is the non-checking counterpart of SETELT. It is only defined when its first operand, reference-vector is a reference vector, and its second operand, s-int, is a small integer and is in the correct range. In that case its value is the same as SETELTs. Otherwise its value is unpredictable, and possible fatal.

MOVEVEC — function

(MOVEVEC vec1 vec2)

Copies the contents of vector vec1 into vector vec2. Both arguments must have the same capacity, and they must be both reference vectors or both binary (character, bit, real or word) vectors.
OPERATIONS ON STRINGS

There are two kinds of strings in YKTLSIP, character strings and bit strings. The elements of a character string are identifiers passing the CHARP predicate. See Figure 21 on page 92. The elements of a bit string are boolean values, either NIL or non-NIL.

Unless otherwise indicated the following operators apply to character strings.

As described in "Character Vectors" on page 30, YKTLSIP strings have both a current length, the content length, and a potential length, the capacity. The capacity is always greater than or equal to the contents. Certain operations will conditionally update a string, depending on its capacity. Thus, RPLACSTR will update its first argument, if the resulting string has a content commensurate with the capacity, and will create a new string otherwise.

CREATION

GETSTR — function

(GETSTR s-int)

Allocates a character vector with a capacity of at least s-int characters. The new vector is returned as the value of GETSTR. Vectors are allocated in increments of full-words; for a character vector, the first word includes only the first character of the string, prefaced by a 3-byte current length field. Therefore, the actual capacity of the vector is defined by

\[((\text{s-int} + 6) / 4) \times 4\) - 3 characters.

<table>
<thead>
<tr>
<th>s-int</th>
<th>Maximum capacity of allocated string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2-5</td>
<td>5</td>
</tr>
<tr>
<td>6-9</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 21. Character string allocation

Zero or negative numbers are invalid values for s-int and will cause an error break. The character string returned by GETSTR is initialized to the null string, that is, its contents length is zero, and it prints as "".

GETFULLSTR — function

(GETFULLSTR s-int [(id | c-str | s-int)])

Similar to GETSTR in that a new character vector is allocated and returned as the value of GETFULLSTR. The new string, however, contains s-int instead of zero characters. The id argument is optional. If it is specified as an identifier, the new string will be initialized so that each character is the initial letter of the P-name of id. If it is a small-integer, the low order eight bits become the fill character. If it is a string, the leftmost character is used. If id is not specified, the string will be initialized to binary zero characters.

GETBITSTR — function

(GETBITSTR s-int)

Operations on strings
Allocates a bit vector with a capacity of at least s-int bits. The new vector is returned as the value of GETBITSTR. Vectors are allocated in increments of full words: for a bit vector, the first word includes only the first 8 bits of the string, prefixed by a 3-byte current length field. Therefore, the actual capacity of the vector is defined by

\[
\left(\left(\frac{s{-}\text{int} + (31 + (3X8))}{32}\times 32\right) - 24\right) \text{ bits}
\]

<table>
<thead>
<tr>
<th>s-int</th>
<th>Maximum capacity of allocated string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-8</td>
<td>8</td>
</tr>
<tr>
<td>9-40</td>
<td>40</td>
</tr>
<tr>
<td>41-72</td>
<td>72</td>
</tr>
</tbody>
</table>

\[\text{Figure 22. Bit string allocation}\]

**STRCONC — function**

\[
(\text{STRCONC} \ c{-}\text{str} \ldots)
\]

This operator returns as its value a new string made by concatenating all the strings \(c{-}\text{str}, \ldots\), \(c{-}\text{str}\) may be either a character vector or a stored identifier (not a GENSYM). In the second case, the print name of the identifier is concatenated into the result string. If \(c{-}\text{str}\) is not a character string or stored identifier, an error is indicated.

**STRINGIMAGE — function**

\[
(\text{STRINGIMAGE} \ \text{item})
\]

This operator creates a character string containing the print representation of \text{item}.

For example, to obtain the character equivalent of an integer, one can write

\[
\begin{align*}
\text{INTEGER} \\
\text{Value} \ &= \ 1352 \\
(\text{STRINGIMAGE} \ \text{INTEGER}) \\
\text{Value} \ &= \ "1352"
\end{align*}
\]

\text{item} may be any LISP value.

**STRINGIZE — function**

\[
(\text{STRINGIZE} \ \text{item})
\]

This operator also computes a string representation for \text{item}. If \text{item} is not a list, the value of \text{STRINGIZE} is \(\text{STRINGIMAGE} \ \text{item}\). If \text{item} is a list, the value of \text{STRINGIZE} is the concatenation of the \text{STRINGIMAGE}s of the elements of \text{item}, with blanks between the elements.

**STRING2BITSTRING — function**

\[
(\text{STRING2BITSTRING} \ c{-}\text{str})
\]

This operator copies its operand, \(c{-}\text{str}\), and converts it into a bit string. The contents of the string are unchanged, only the type of the pointer to it, and the contents length are modified.

This operator allows direct access to the bit pattern of a character string.

**BITSTRING2STRING — function**

\[
(\text{BITSTRING2STRING} \ b{-}\text{str})
\]
This operator copies its operand, b-str, and converts it into a character string. The contents of the string are unchanged, only the type of the pointer to it, and the contents length are modified.

PNAME — function

(PNAME id)

Returns a copy of the print name of id. If the value of id is not an identifier, an error break is taken. The print name is a character string.

While PNAME is primarily thought of as an operator on identifiers, it is also a creation operator for strings.

ANDBIT — function

(ANDBIT b-str ...)

ANDBs b-str ... and returns as value the resultant string. None of the argument strings is changed by the operation.

An error is indicated if any operand is not a bit string or if the strings are not equal in length.

ORBIT — function

(ORBIT b-str ...)

ORs b-str ... and returns as value the resultant string. None of the argument strings is changed by the operation.

An error is indicated if any operand is not a bit string or if the strings are not equal in length.

XORBIT — function

(XORBIT b-str ...)

Exclusive ORs b-str ... and returns as value the resultant string. None of the argument strings is changed by the operation.

An error is indicated if any operand is not a bit string or if the strings are not equal in length.

MAKETRITABLE — function

(MAKeTRITABLE table-definition item)

This operator creates a V/370 translate-and-test table. Operands are a collection of characters-of-interest, table-definition and an inversion flag, item. The first operand can be either a string or a list. If a string it is converted to a list of identifiers using VEC2LIST.

The list elements may be small integers, strings or non-GENSYM identifiers, or pairs with one of the above as CAR and a small-integer as CDR. The table, a newly created character string of length 256, is initialize with x00 (if item is NIL), or xFF (if item is -NIL). Then the positions corresponding to the list elements (or their CARs, if pairs) are set to xFF or x00 (opposite of the initial value), or to the CDR value for pairs.

The resulting string is a suitable operand for STRPOS, STRPOSL, and STRTRT.

MAKESTRING — function

(MAKESTRING c-str)
This operator returns a copy of its argument. Its primary use is in compiled code, where it generates code to produce a string from data stored in the bpi, rather than in the data heap. This is done to free the more valuable, garbage collected, space.

ACCESSING

STRINGLENGTH — function

(STRINGLENGTH str)

This operator returns as its value the current length of the character or bit string STR. An error is indicated if STR is not a character or bit string.

As with non-string vectors, the size of a string is an intrinsic part of the object. Again, note that this is the size of the contents of the string, not its capacity. As of now, there is no operator which will report the capacity of a string directly, but see LENGTHCODE, following.

QSTRINGLENGTH — function

(QSTRINGLENGTH c-str)

This is the non-checking counter part of STRINGLENGTH. If C-STR is a character string, it returns its contents length. If C-STR is not a character string its result is unpredictable.

LENGTHCODE — function

(LENGTHCODE str)

LENGTHCODE returns as its value the size, in bytes, of the string STR. If VEC is a character or bit vector, this value is the maximum size specified, including the 3-byte current string length field.

To determine the capacity of a string, LENGTHCODE must be used. For a character string the capacity is

(DIFFERENCE (LENGTHCODE c-str) 3)

For a bit string it is

(TIMES 8 (DIFFERENCE (LENGTHCODE b-str) 3))

QLENGTHCODE — function

(QLENGTHCODE str)

This is the non-checking counter part of LENGTHCODE. If STR is a string, it returns its contents length. If STR is not a string its result is unpredictable.

See also page undefined volength...

FETCHCHAR — function + compiler macro

(FETCHCHAR c-str s-int)

Returns as value the character object (identifier whose print name is a single character) corresponding to the s-intth element of the character string C-STR. An error is indicated if C-STR is not a string, or if S-INT is negative or exceeds the current length of the string C-STR.

ELT — function + compiler macro

(ELT str s-int)
For character strings,

(ELT c-str s-int)

is exactly equivalent to

(FETCHCHAR c-str s-int)

For bit strings ELT returns a value of NIL or #T. ELT is the only access operator for bit strings.

SUBSTRING — function + compiler macro

(SUBSTRING c-str s-int1 s-int2)

This macro returns a copy of part (or all) of c-str. The returned value starts with the s-int1 (index) character of c-str (remember, index zero is the first character) and is s-int2 (length) characters long. If s-int2 is specified as (), that designates the end of the string.

STRING2ID-N — function

(STRING2ID-N c-str s-int)

This operator treats its first operand, c-str, as a collection of tokens, where the tokens are substrings, separated by one or more blanks. (Leading blanks are ignored.) The s-int-th token is extracted and INTERNed (see page 113) and the resulting identifier is returned as the value. If there are less than s-int tokens in c-str, the value is zero (0).

STRING2INT-N — function

(STRING2INT-N c-str s-int)

This operator treats its first operand, c-str, as a collection of tokens, where the tokens are substrings, separated by one or more blanks. (Leading blanks are ignored.) The s-int-th token is extracted and, if it consists wholly of digits, the corresponding positive small integer is returned as the value. If there are less than s-int tokens in c-str or the s-int-th token contains non-digit characters, the value is NIL.

VEC2LIST — function

(VEC2LIST str)

This operator constructs a new list, containing the elements of the string, str. The elements of the resulting list will be identifiers, in the case of character strings, and (NIL | #T) in the case of bit strings.

SEARCHING

STRPOS — function

(STRPOS c-str1 c-str2 s-int item)

This operator searches c-str2 for a substring, c-str1, which may contain don't care characters. S-int is the an index, indicating the starting position for the search. Item is NIL or the don't care character. An error break is taken if item is not a small integer, identifier, string or NIL.

The value is NIL if the substring is not found, and the index of the first character of the sub-string.
STRPOSL — function

(STRPOSL table c-str s-int item)

This operand searches a string for a character from a given set. Table should be a translate-and-test table, as built by MAKEITRTABLE, or a valid first argument for that function. C-str is the string to be searched. S-int is an index, indicating the point in the string at which the search is to start. Item is a flag; if NIL search ends at first character specified by table, otherwise search ends at first character not specified by table. The value is NIL, if the search fails, or the index of the found character.

X
Value = 'This is a test of those operations.'
(STRPOSL 'aeiou' X 0 "I")
Value = 2
(STRPOSL 'aeiou' X 0 NIL)
Value = 0
(STRPOSL 'aeiou' X 3 "T")
Value = 5
(STRPOSL 'aeiou' X 6 "T")
Value = 0

STRRT — function

(STRRT table c-str pair)

This operator searches c-str for a specified character or characters. It is similar to STRPOSL, but without the negation flag, and the position argument (PAIR) is an updatable pair, whose CAR is the starting position.

If the desired character is not found the value is NIL. If it is found, the value is the pair is updated, with its CAR being set to the position of the found character and its CDR being set to the table entry. If pair is not a pair, one is created and initialized with zero.

X
Value = 'This is a test of those operations.'
(SETQ Y (CONS 0 0))
Value = (0 . 0)
(STRPOSL "(a . 1) (a . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (2 . 3)
(RPLACA Y (ADDI (CAR Y)))
Value = (3 . 3)
(STRPOSL "(a . 1) (a . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (5 . 3)
(RPLACA Y (ADDI (CAR Y)))
Value = (6 . 3)
(STRPOSL "(a . 1) (a . 2) (i . 3) (o . 4) (u . 5)) X Y)
Value = (8 . 1)

Note that in the above example it would be more efficient to call MAKEITRTABLE once with the first operand, and then pass that as the value to STRPOSL. The test in STRPOSL to see if a ready made RT table has been provided is a heuristic one. If the first operand is a string containing 256 characters, it is used as a RT table, otherwise it is passed to MAKEITRTABLE.
UPDATING

CHANGELENGTH — function + compiler macro

(CHANGELENGTH str s-int)

The length of the string str is updated to be the value s-int. An error is indicated if str is not a character or bit string, or if the value of s-int exceeds the maximum potential length of str. The value of CHANGELENGTH is the string str with its new length.

QSCANGELENGTH — function + compiler macro

(QSCANGELENGTH c-str s-int)

This is the non-checking counterpart to CHANGELENGTH. If c-str is a character string, and s-int is a small integer ≤ the capacity of c-str, then the result is equivalent to CHANGELENGTH. Otherwise the result is unpredictable, and possible fatal.

TRIMSTRING — function

(TRIMSTRING c-str)

This operator updates the lengthcode in its argument, to produce a string with the minimum capacity which will hold the current contents. The portion of the string which is discarded (if any) becomes inaccessible.

STORECHAR — function + compiler macro

(STORECHAR c-str s-int id)

Updates c-str by replacing the s-intth character with the first character of the print name of the stored identifier id. An error break is taken if c-str is not a character string, or if s-int is negative or exceeds the current length of the string c-str, or if id is not a stored identifier.

SUFX — function + compiler macro

(SUFFIX id c-str)

Updates the character string c-str by adding the first character of the print name of the stored identifier id to the end of the string. id is usually a character object, but may be any stored identifier. This function increments the length of the character string by one, or causes an error break if there is not sufficient space in the string c-str for the additional character.

The value of STORECHAR is the last argument to STORECHAR, the value used to update the designated character of the string.

SETLET — function + compiler macro

(SETLET str s-int item)

This is the inverse function of ELT -- it updates the s-int element of str to be item. For character strings, item must be an identifier. For bit strings it may have any value, with NIL and non-NIL being distinguished.

SETLET will take an error break if s-int is out of range, or if item is not compatible with the type of str.

The value of SETLET is item, the value used in updating the specified object.

RPLACSTR — function + compiler macro

(RPLACSTR c-str1 s-int1 s-int2 c-str2 [s-int3 [s-int4]])
This is a generalized string modification routine. It can replace any part of C-STR1 with any part of C-STR2, making any necessary adjustment in the length of C-STR1 because the replacement characters from C-STR2 are greater or fewer than the characters being replaced in C-STR1. Furthermore, it can insert C-STR2, or some specified substring of C-STR2, into C-STR1.

C-STR1 must be a character vector, else an error is indicated. C-STR2 may be either a character vector, or it may be a stored identifier (not a GENSYM). In the latter case, the print name of the identifier (which is a character string) will be used as C-STR2, and s-int3 and s-int4 refer to this string. If C-STR1 or C-STR2 are not as described, an error is indicated.

S-int1 specifies the index of the first character in C-STR1 to be replaced. S-int2 specifies the number of consecutive characters, beginning with the S-int1 character, to be replaced. S-int3 and S-int4 specify the location and number of characters from C-STR2 which are to replace the designated characters in C-STR1.

S-int1, S-int3, S-int2 and S-int4 may be either integer values or NIL; an error is indicated if they are not.

In general, an INDEX may vary from zero to the current length-1. If NIL is specified for an INDEX, the numeric value zero is used. If S-int1 is equal to the current length, S-int2 must be zero; by use of this convention, C-STR2 can be appended to the end of C-STR1.

If zero is specified for S-int2, C-STR2 is inserted in C-STR1 before the position specified by S-int1. If NIL is specified for a length, all of the characters from the related index value to the end of the string are used. In effect, using NIL for the value of LENX is an efficient way of specifying the value:

(DIFFERENCE (STRINGLENGTH STRx) INDEXx)

S-int4 and S-int3 are optional arguments. If they are not specified, a value of NIL will be assumed.

Whenever possible, RPLACSTR will update the original C-STR1 and return as its value the updated string. However, if S-int4 is greater than S-int2, it is possible that C-STR1 does not have sufficient space for the result string. In this case, a new string is constructed and this new string is returned as the value of RPLACSTR.

The user may test whether the updated string is the original C-STR1 or a copy by an expression such as:

(EQ C-STR1 (SETQ TEMP (RPLACSTR C-STR1 ... )))

which will be true if C-STR1 has been updated in place, and false if a new string had to be created. The purpose of the SETQ operation is to preserve the value of RPLACSTR in case a new string was created.

L-CASE — function

(L-CASE c-str)

This operator, when applied to a character string, translates all lower-case alphabetic character to their upper-case equivalents. This translation is done in place, updating the operand.

L-CASE will also accept a list of character strings, in which case its value is a new list of the translated strings.

U-CASE — function

(U-CASE c-str)
This operator, when applied to a character string, translates all upper-case alphabetic character to their lower-case equivalents. This translation is done in place, updating the operand. U-CASE will also accept a list of character strings, in which case its value is a new list of the translated strings.

COMPARING

STRGREATERP — function

(STRGREATERP C-STR1 C-STR2)

This function compares two character strings and returns true if C-STR1 is greater than C-STR2, otherwise it returns NIL. The comparison is done using the S/370 CLCL instruction.

If the two strings are of unequal length, the shorter string is considered to be padded on the right with binary zeros for purposes of comparison. If an argument is not a character string, an error break occurs.

BITGREATERP — function

(BITGREATERP B-STR1 B-STRN)

Compares two bit strings, and returns true if B-STR1 is greater than B-STRN. If the strings are unequal in length, the shorter string is considered to be padded on the right with zeros for purposes of comparison. The argument strings are not changed by this function, even the bits beyond the current length of the strings are preserved.

If B-STR1 or B-STRN is not a bit string, an error break occurs.
The majority of YKLISP operators which expect numbers as operands will accept either fixed or floating arguments. Unless otherwise stated, any operand receiving mixed operands, will produce a floating value. When the value of an operation is an integer, it will be represented by either a small integer or a large integer, irrespective of the form of its arguments. See "Numbers" on page 23.

In a sense there are only number creating operators. Since numbers are non-updatable objects, each numeric value is a new number.

CONVERSION.

FLOAT — function

(FLOAT num)

If num is an integer number, the value is the closest real approximation to that number, unless num exceeds the range of floating point numbers (approximately $10^{75}$), in which case an error break is taken. An error break occurs also if num is not numeric. If num is already a real number, it is returned as the value of FLOAT.

FIX — function

(FIX num)

If num is a real (floating point) number, returns the integral part of that value as an integer. If num is already an integer number, it is returned as the value of FIX. For other values of num, an error break is taken.

PREDICATES

PLUSP — function

(PLUSP item)

Returns item if item is positive or zero, NIL if item is negative. If item is non-numeric the value is NIL.

QPLUSP — function + compiler macro

(QPLUSP s-int)

The non-checking counterpart of PLUSP. If s-int is not a small integer the value is unpredictable.

The operators whose names start with QS (with the exception of QSORT) are only defined for small integer arguments. The computational QS... operators always return small integer results. Certain of these operators have further restrictions, which are detailed in their definitions.

ZEROP — function

(ZEROP item)

Returns item if item is zero, NIL if item is non-zero. If item is non-numeric the value is NIL.

QZERO — function + compiler macro

(QZERO s-int)
The non-checking counterpart of ZEROP. If s-int is not a small integer the value is unpredictable.

**MINUSP** — function

(MINUSP item)

Returns NIL if item is positive or zero, item if item is negative. If item is non-numeric the value is NIL.

**QSMINUSP** — function + compiler macro

(QSMINUSP s-int)

The non-checking counterpart of MINUSP. If s-int is not a small integer the value is unpredictable.

**ODDP** — function

(ODDP item)

Returns NIL if item is even, item if item is odd. If item is not a fixed number the value is NIL.

**QSODDP** — function + compiler macro

(QSODDP s-int)

The non-checking counterpart of ODDP. If s-int is not a small integer the value is unpredictable.

**GREATERP** — function

(GREATERP num1 num2)

For num1 and num2 numeric values, compares them and returns T if num1 is greater than num2, or NIL if num1 is not greater than num2. Should either num1 or num2 be a floating point value, the real fuzz factor may affect the comparison. There is a discussion of this in the section on data types, page 29. Basically, the fuzz factor allows two real values which are close in value to be considered equal, thus neither is greater than the other. The value of the real fuzz factor determines what close means.

**QSGREATERP** — function + compiler macro

(QSGREATERP s-int1 s-int2)

The non-checking counterpart of GREATERP. If s-int1 and s-int2 are not small integers the value is unpredictable.

**LESP** — function

(LESP num1 num2)

The value of (LESP X Y) is equivalent to the value of (GREATERP Y X), however the order of evaluation of the operands, and hence of side effects, is maintained.

**QSELSSP** — function + compiler macro

(QSELSSP s-int1 s-int2)

The non-checking counterpart of LESSP. If s-int1 and s-int2 are not small integers the value is unpredictable.

**COMPUTATION**

**MAX** — function + compiler macro

(MAX num ...)

106 YKTLISP Program Description and Operations Manual
The value of this operator is the algebraically largest argument value. If any of the arguments is not a number, an error break occurs.

**QSMAX** — function + compiler macro

(QSMAX s-int1 s-int2)

The non-checking counterpart of MAX, defined for two small integer operands. If s-int1 and s-int2 are not small integers the value is unpredictable.

**MIN** — function + compiler macro

(MIN num ...)

The value of this operator is the algebraically smallest argument value. If any of the arguments is not a number, an error break occurs.

**QSMIN** — function + compiler macro

(QSMIN s-int1 s-int2)

The non-checking counterpart of MIN, defined for two small integer operands. If s-int1 and s-int2 are not small integers the value is unpredictable.

**ABSVAL** — function

(ABSVAL num)

Returns the absolute value of NUM, if X is a number. If num is not a number, an error break is taken.

**QSABSVAL** — function + compiler macro

(QSAABSVAL s-int)

Returns the absolute value of NUM, if X is a number. If s-int is not a small integer the value is unpredictable.

**MINUS** — function

(MINUS num)

Unary minus operation. NUM may be any number. Value returned is minus X. If NUM is non-numeric an error break is taken.

**QSMINUS** — function + compiler macro

(QSMINUS s-int)

The non-checking counterpart of MINUS. If s-int is not a small integer the value is unpredictable.

**PLUS** — function + compiler macro

(PLUS num ...)

This operator computes the sum of NUM ..., where those arguments are numeric values. If any of NUM are not numeric, an error break is taken. If all of NUM are integers, the result is an integer. Otherwise, the result is real.

**QSPPLUS** — function + compiler macro

(QSPPLUS s-int1 s-int2)

The non-checking counterpart of PLUS, defined for two operands, both of which are small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.
ADD1 — function

(ADD1 X)

Returns as its value (PLUS X 1). If X is not numeric an error break will be taken.

QSADD1 — function + compiler macro

(QSADD1 s-int)

The non-checking counterpart of ADD1. If s-int is not a small integer the value is unpredictable.

QSINCl — function + compiler macro

(QSINCl s-int)

This operator is equivalent to QSADD1, except when its operand is `-1`. In that case its value is unpredictable, and possibly fatal.

QSINCl compiles to a single machine instruction, and is used for index arithmetic in many internal functions.

DIFFERENCE — function + compiler macro

(DIFFERENCE num1 num2)

This function computes the value num1 minus num2, where num1 and num2 are numeric values. If either num1 or num2 is not numeric, an error break is taken. If both num1 and num2 are integers, the value of DIFFERENCE will be an integer. Otherwise, the result value is real.

QSDIFFERENCES — function + compiler macro

(QSDIFFERENCES s-int1 s-int2)

The non-checking counterpart of DIFFERENCES, defined for small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.

SUBLI — function

(SUBLI X)

Returns as its value (DIFFERENCE X 1). If X is not numeric an error break will be taken.

QSSUBLI — function + compiler macro

(QSSUBLI s-int)

The non-checking counterpart of SUBLI. If s-int is not a small integer the value is unpredictable.

QSDECLI — function + compiler macro

(QSDECLI s-int)

This operator is equivalent to QSSUBLI, except when its operand is 0. In that case its value is unpredictable, and possibly fatal.

QSDECLI compiles to a single machine instruction, and is used for index arithmetic in many internal functions.

TIMES — function + compiler macro

(TIMES num ...)

108 YKTLISP Program Description and Operations Manual
This operator computes the product of NUM ..., where these arguments are numeric values. If any of NUM are not numeric, an error break is taken. If all of NUM are integers, the result is an integer. Otherwise, the result is real.

QSTIMES — function + compiler macro
(QSTIMES s-intl s-int2)
The non-checking counterpart of TIMES, defined for two operands, both of which are small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.

DIVIDE — function + compiler macro
(DIVIDE num1 num2)
DIVIDE computes the quotient of num1 divided by num2, and returns as value a list:

(QUOTIENT REMAINDER).
The first element of this list is the quotient and the second element is the remainder. If both num1 and num2 are integers, the quotient and remainder will be integers. If any argument is real, the quotient and remainder will be real. The remainder for a real quotient is peculiar because it exists only because of the approximation needed for representing real numbers as floating point numbers in the computer. A real remainder is computed as

(DIFFERENCE num1 (TIMES num2 QUOTIENT)).

If the divisor is zero, or if either argument is non-numeric, and error break is taken

QUOTIENT — function + compiler macro
(QUOTIENT num1 num2)
This function is similar to DIVIDE in that it computes the quotient of num1 divided by num2, but differs in that it returns only the quotient, rather than a list of quotient and remainder. If num1 and num2 are both integers, the quotient will be an integer. Otherwise, the quotient is real.

If the divisor is zero, or if either argument is non-numeric, and error break is taken

QSTQUOTIENT — function + compiler macro
(QSTQUOTIENT s-intl s-int2)
The non-checking counterpart of QUOTIENT, defined for small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.

REMAINDER — function + compiler macro
(REMAINDER num1 num2)
Returns as value the remainder of num1 divided by num2. If both num1 and num2 are integers, the remainder is computed from an integer division. If either num1 or num2 is floating point, the remainder is computed by subtracting num2 times the real quotient from num1. See DIVIDE, this page.

If the divisor is zero, or if either argument is non-numeric, and error break is taken

QSTREMAINDER — function + compiler macro

Operations on numbers.
(QSRREMAINDER s-int1 s-int2)

The non-checking counterpart of REMAINDER, defined for small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.

LEFTSHIFT — function

(LEFTSHIFT num s-int)

If NUM is a small integer or one word large integer, the value of this function is the number (limited to the range of a one word large integer) obtained by a binary shift of S-INT bits. Positive values of S-INT denote a left shift, negative values a right shift. Any bits shifted outside of a 32 bit word are lost, and zero bits are supplied as needed. If NUM is not within the described range, an error break is taken.

QSLLEFTSHIFT — function + compiler macro

(QSLLEFTSHIFT s-int1 s-int2)

The non-checking counterpart of LEFTSHIFT, defined for small integers. If either of the arguments is not a small integer, or if the result is outside the small integer range, the value is unpredictable, but will be a small integer.

RIGHTSHIFT — function + compiler macro

(RIGHTSHIFT num s-int)

Equivalent to (LEFTSHIFT num (MINUS s-int)).

ALINE — function

(ALINE s-int1 s-int2)

Value is the small integer S-INT1 rounded up to the nearest multiple of the small integer S-INT2, where S-INT2 is a power of 2. If S-INT1 or S-INT2 are not small integers or are negative, an error break is taken. If S-INT1 is zero or one, returns S-INT1 unchanged. If S-INT2 is not a power of 2, result is unpredictable.

(ALINE 15 4) = 16.

QSNOT — function + compiler macro

(QSNOT s-int)

Returns the bitwise complement of the numeric value of S-INT, as a small integer. If S-INT is not a small integer the value is unpredictable, but will be a small integer.

QSAND — function + compiler macro

(QSAND s-int1 s-int2)

Returns the bitwise AND of the numeric values of S-INT1 and S-INT2, as a small integer. If S-INT1 and S-INT2 are not small integers the value is unpredictable, but will be a small integer.

QSOR — function + compiler macro

(QSOR s-int1 s-int2)

Returns the bitwise OR of the numeric values of S-INT1 and S-INT2, as a small integer. If S-INT1 and S-INT2 are not small integers the value is unpredictable, but will be a small integer.

QSXOR — function + compiler macro
(QSXOR s-int1 s-int2)

Returns the bitwise exclusive OR of the numeric values of s-int1 and s-int2, as a small integer. If s-int1 and s-int2 are not small integers the value is unpredictable, but will be a small integer.

EXPT — function

(EXPT num1 num2)

Returns the value of num1 raised to the num2 power. The value is an integer if num1 is an integer and num2 is a positive integer; otherwise, the value is a floating point number1.

num1 cannot be negative if num2 is not a positive integer.

EXP — function

(EXP num)

Returns the value of e raised to the num power.

An error break will be taken if num > 174.66.

LN — function

(LN num)

This operator computes the natural logarithm of num. num may be either integer or floating, but it must be within the range of a floating number. The result is a floating number.

LOG — function

(LOG num)

Computes common (base ten) logarithm of num. num may be either integer or floating, but must be within the range of a floating number. The value is a floating point number.

LOG2 — function

(LOG2 num)

Computes the logarithm to the base 2 of num. num may be either integer or floating, but must be within the range of a floating number. The value is a floating point number.

SIN — function

(SIN num)

Computes the sine of num, the angle in radians. num may be either integer or floating, but must be < approximately 3.5x10^13. The value is a floating point number.

COS — function

(COS num)

Computes the cosine of num, the angle in radians. num may be either integer or floating, but must be < approximately 3.5x10^13. The value is a floating point number.
OPERATIONS ON IDENTIFIERS

In YKLISP identifiers are of two types, normal or stored, and GENSYM or non-stored.

GENSYM identifiers have only a single component, their pname, which has the form 'GN', with 1 ≤ n ≤ 2^4. They can be used as lexical or global variables, but not as fluid variables.

GENSYM identifiers are treated specially by the standard PRINT and READ routines, in that they are identifiable as GENSYM's in printed output and when one is read, it is replaced by a new GENSYM in the structure created by the READ program. Thus, if the same expression is read several times, it will contain unique GENSYM's in each copy read, although there will be only one new GENSYM created for each distinct GENSYM in the expression being read. If the same GENSYM occurs more than one time in the input expression, the same newly created replacement GENSYM will be referenced every place the original GENSYM was referenced.

X
Value = %G1325

Y
Value = %G1325
(EQ X Y)

Value = NTL

(SETQ A "%G1")

Value = %G1401

(SETQ B "%G1")

Value = %G1402

(EQ A B)

Value = NIL

(SETQ C "(G1 %G2 %G1 %G2))")

Value = (%G1403 %G1405 %G1403 %G1405)

There is only a finite number of possible GENSYMS, .75×2^4, and an error break is taken if an attempt is made to generate more than that many.

There are two types of normal identifiers, in turn.

INTERNed identifiers are remembered in a system table, the OBARRAY (sometimes referred to as the OBLIST, for historical reasons). Two identifiers with identical pnames will always be EQ, even when read on different occasions.

UNINTERNed identifiers can only be created internally, they can not be read. Different applications of UNINTERN to the same pname result in different, non-EQ, objects. An UNINTERNed identifier is not distinguishable upon printing and if read in from a file, will be interpreted as an INTERNED identifier. UNINTERNed identifiers are meant to be used internally, where there is a danger of exhausting the set of GENSYMS.

Normal identifiers have two components, their pname and their property list.

CREATION

INTERN — function

(INTERN c-str)

This operator searches the object array for a pre-existing identifier with a pname EQUAL to c-str. If one exists, it is returned as the value of INTERN. Otherwise, a new identifier is created, with a print name EQUAL to c-str and added to the object array. This new identifier is returned as the value of INTERN. If c-str is not a character string, an error break occurs.

UNINTERN — function

Operations on identifiers 113
(UNINTERN C-str)

This operator creates a new identifier, with a pname EQUAL to C-str. This new identifier is returned as the value of UNINTERN. It is not added to the object array. If C-str is not a character string, an error break occurs.

GENSYM — function

(GENSYM)

This operator constructs a new, unique, non-stored identifier.

The mechanism used to insure unique ID's is simply to have a counter which is incremented every time a new GENSYM is required and to incorporate this counter's value into the print name of the identifier.

Because there are a limited number of GENSYM's, it is recommended that GENLABEL's be used whenever they are appropriate rather than the more costly GENSYM's.

GENLABEL — function

(GENLABEL)

The value of this operator is a non-stored constant suitable for use as a statement label inside of a PROG expression or in a LAP contour. These objects pass the GENSYMP predicate. These constants are generated in a series which is reset to its starting value by the DEFINATE function so that the same values may be reused. The principal use of GENLABEL is by macro definitions which require a locally unique label to be incorporated into their expansion.

The only difference between the value of GENLABEL and GENSYM is the range of numeric values used. They are in all other respects simply GENSYM identifiers.

GENLABEL's share the peculiar nature of GENSYM's with respect to READ. They are never read in verbatim, but rather each distinct GENLABEL in an expression being read is replaced in the new structure READ produces by a newly generated GENSYM.

L-CASE — function

(L-CASE id)

The L-CASE operator returns the identifier whose pname is the result of translating any upper case alphabetic characters in the pname of id into their lower case equivalents.

See page 112 for the behavior of U-CASE when applied to strings.

U-CASE — function

(U-CASE id)

The U-CASE operator returns the identifier whose pname is the result of translating any lower case alphabetic characters in the pname of id into their upper case equivalents.

See page 102 for the behavior of U-CASE when applied to strings.

ACCESSING

PHNAME — function

(PHNAME id)

114  YKTLISP Program Description and Operations Manual
Returns a copy of the print name of id. If the value of id is not an identifier, an error break is taken. The print name is a character string.

GET — function

(GET id item).

See page 86 for the description of the behavior of GET when applied to lists. If id is not an identifier or a pair, value is NIL. Otherwise the property list of id is searched for the first occurrence of an element such that

(EQ item (CAR element))

is true. If found, the value of GET is (CDR element). If such an element is not found, the value of GET is NIL.

PROPLIST — function

(PROPLIST id)

The value of id must be an identifier. Returns the property list associated with that identifier. The property list is a standard LISP association list. The value returned is not the actual property list, but is the result of applying APPEND to the property list and NIL. Thus, while the name-value pairs may be updated, the actual property list itself is secure.

SEARCHING AND UPDATING

MAKEPROP — function

(MAKEPROP id item1 item2)

Operator to update the property list of the identifier id. If the item1 property already exists, its associated value is changed to item2. If the item1 property does not currently exist, a new property with this name is put at the beginning of the property list.

The value of id must be an identifier or a list, but item1 and item2 may be any expressions. For a description of the behavior of MAKEPROP when applied to lists, see page 87.

DEFLIST — function

(DEFLIST list item)

This function expects list to be a list of pairs whose CARs are identifiers and whose CADRs are arbitrary values. For each of these pairs, (MAKEPROP id item VALUE) is performed, assigning the CDR value from the pair as the value of the item property in the identifier's property list.

The value of DEFLIST is a new list containing the property values (the CADRs of the elements of list).

To put NUM properties on each of the identifiers A through F, with values 10 through 15, one would write

(DEFLIST "A" 10 "B" 11 "C" 12 "D" 13 "E" 14 "F" 15)

"NUM"

Value = (10 11 12 13 14 15)

REMPROP — function

(REMPROP id item)
The item property of the identifier id is removed from the property list of id. The value of REMPROP is NIL if there is no item property. If the property exists, the value of REMPROP is the value associated with that property.

UPDATING

REMAILPROPS — function
(REMAILPROPS id)

All properties on the property list of id are removed. If id is not a normal identifier the value of REMMAILPROPS is NIL, otherwise the value is id.

OBJECT ARRAY

OBARRAY — function
(OBARRAY)

Returns as value a copy of the current LISP object array. This is a reference vector containing elements which are identifiers (INTERN’ed variables).

Because the value of OBARRAY is a copy of the actual object array, it may be modified in any way by the user.
YKTILISP supports two major styles of I/O, stream I/O to and from disk files and terminals, and key addressed I/O to specially formatted disk files. This section covers the former, while the "key addressed I/O" on page 125 covers the latter.

At its simplest, a stream can be considered a source of, or recipient of a sequence of characters. In practice, the essentially line (or record) oriented devices show through to a greater or lesser extent. See "Streams" on page 33.

YKTILISP provides six pre-constructed streams. These are:

CURINSTREAM — variable
The currently active input stream. Each invocation of SUPV results in a new binding of this variable. This may be either a console (terminal) stream or a file (disk) stream.

CUROUTSTREAM — variable
The currently active output stream. This may be either a console stream or a file stream. Also bound by SUPV.

ERRORINSTREAM — variable
The input stream used by the break loop. Normally a console stream. Distinguished from CURINSTREAM in case an error is detected attempting to read from CURINSTREAM.

ERROROUTSTREAM — variable
The output stream used by the break loop. Normally a console stream.

NULLOUTSTREAM — variable
A data sink. This stream simply discards anything written to it.

STACKLIFO — variable
An output stream which places lines in the console stack, so that they will be seen by any program reading from the console.

There are really two classes of operators described in this section, those which use streams to perform IO and those which operate on the streams as data objects.

CREATION

DEFDSTREAM — function
(DEFDSTREAM list s-int1 s-int2)
This function constructs a standard structure usable as an input/output stream by the normal READ and PRINT programs. The structure which is created and returned as the value of DEFDSTREAM is not checked for validity by DEFDSTREAM. The first use of the stream will typically involve initialization and verification of the data in the structure according to the needs of the using function. Thus, DEFDSTREAM will create an input stream for a non-existent file, and no error will be indicated until an attempt is made to read from the resulting stream.

List is an association list which defines some of the characteristics of the stream being created. S-int1 is an integer defining the length of the lines for which buffer space is to be
provided. This represents a maximum length; shorter lines may be produced by using TERPRI for output, and shorter lines may be omitted by whatever source an input stream uses. A line buffer is not allocated by DEFIODSTREAM, but will be allocated the first time the stream is used by READ or PRINT. $<int>$ designates a particular record within a data set. A value of zero means the first record if an input stream, or the next record if an output stream.

Streams may be defined by this function based either on a disk file or console as an input/output device. The list value is examined to determine which of these devices is to be used, and an appropriate program is selected and stored as the value of rfnn in the stream structure.

In order to implement commonly-needed default stream attributes, DEFIODSTREAM makes the following modifications to list:

- If no MODE property is already part of list, (MODE , INPUT) is added to list by nondestructive CONSing.
- If (MODE , I) or (MODE , O) is specified, I or O is REPLACed by INPUT or OUTPUT, respectively.

Following is a description of the properties and meanings which are most commonly used for list. Additional properties are ignored by the standard stream processing functions, but may be added to provide additional information to be used by the user's programs.

- (DEVICE , CONSOLE) is specified to indicate this stream is defined on the user's console.
- (FILE filename filetype) or (FILE filename filemode) is specified to indicate this stream is defined on the designated disk file. Values for filename and filetype may be specified either as identifiers or character strings.
- (MODE , INPUT) or (MODE , OUTPUT) designates an input or output stream, respectively. (This attribute was discussed above.)
- (RECFM , V) or (RECFM , F) designates whether a disk file referenced in an output stream is to have fixed or varying length records in it. Varying length records is the default assumption.
- (QUAL , {S | T | U | V | X}) designates the type of CMS read operation to be used in obtaining records from the console for this stream. The letters have the following meanings:
  - S = pad records with blanks to 120 characters.
  - T = read a logical line (the default operation).
  - U = pad with blanks and translate to upper case.
  - V = translate to upper case.
  - X = read a physical line.
- (QUAL , {LIFO | FIFO | NOEDIT}) specifies for console output files that no editing is to be performed on output lines (i.e. for typewriter consoles, trailing blanks are not deleted and a carriage return is not automatically appended to the line). LIFO and FIFO designate that output lines are to be placed into the console input stack, rather than be written to the console.

SHUT — function

<SHUT strm>

This operator invokes a system-dependent routine to close any file related to the argument strm. If no file is actually in need of closing, the action of SHUT is effectively a no-operation. The value of SHUT is -1 if strm is ill formed, 0
if the stream is successfully shut or un-shut-able (a console stream, for example), and the return code from the system dependent shut routine otherwise.

Note that, for an output stream, if a partial buffer of data exists, an explicite TERPRI must be done before SHUT is applied, otherwise the data will be lost.

INPUT

end-of-file — concept

It is possible for a stream to become empty. This occurs, for example, if a DASD file reaches end-of-file.

Unfortunately, when using READ, it is impossible to reliably tell that you have just read the last item from a stream. Thus, any use of READ must be prepared to receive a read-place-holder as the value of a use of that operator. Read-place-holders are special, non-representable, objects which can be identified by the operator PLACE (page undefined read-place), and which are only encountered as the value of READ and READ-LINE when an attempt is made to read beyond the end of a stream.

NEXT — function + compiler macro

(NEXT strm)

This operator advances strm to the next character and returns the updated strm as its value; the CAR of the strm is this next character.

If the last character in the buffer of strm has been extracted, it is put into the end-of-list configuration. If it is in the end-of-list configuration, an attempt is made to re-fill the buffer. If the buffer is refilled the first character is extracted from it, otherwise the stream is put into the empty configuration. (See "Streams" on page 33).

NEXT does not check strm to verify that it is an input stream.

ITEM-N-ADV — function + compiler macro

(ITEM-N-ADV strm)

This operator extracts the current object at the head of the stream strm, and then advances strm, using NEXT. Its value is the object extracted, i.e., that which was current at the time it was invoked, rather that that which is current at the time it returns. Returns as value the current object at the head of the stream strm, then advances strm using NEXT. This function discards line and indications, so the caller will see only actual data values from strm.

RDCHR — function + compiler macro

(RDCHR [strm])

This operator is equivalent to ITEM-N-ADV.

The operand, strm, is optional, and defaults to CURINSTREAM if omitted.

READ-LINE — function

(READ-LINE strm)

This operator reads one line from strm. This is a single record, from a DASD file, or the contents of the input area, for a console stream.
If `strm` is in the end-of-file condition the value is a read-place-holder, otherwise it is a string.

READ — function + compiler macro

(READ [strm])

This operator reads one complete list object from `strm`.

If `strm` is omitted it defaults to CURINSTREAM.

READ references three free variables which control the parsing of input data. These are QUOTEIZER, STRINGIZER and LETTERIZER, which are initially set to ",", '" and I, respectively. Each should be bound to the character object which is to signal the READ program to perform an appropriate operation. The QUOTEIZER character indicates that the symbolic expression immediately following (there may be no intervening blanks before the first character of this expression) is to become the second element in a list whose first element is the identifier QUOTE.

"(A B C)"

is equivalent to

(QUOTE (A B C))

This is a particularly useful facility when typing expressions interactively from a terminal.

The STRINGIZER character is simply the character designated to act as a string delimiter.

The LETTERIZER is the character used to signal that the immediately following character is to be considered a data character rather than a control character. The LETTERIZER character is needed for designating, for example, the identifier having the print name 999, or the character object left parenthesis or blank.

\begin{verbatim}
[999
Value = 999
(LENGTH "(( | ))")
Value = 3
\end{verbatim}

When a stream is empty (at end-of-file), the value of READ is a read place holder. See the discussion under PLACEP, page 74.

When a stream becomes empty it is put into a peculiar configuration, which unfortunately discard all useful information, such as file name. A stream is empty when:

\begin{verbatim}
(EQ strm (CAR strm))
(EQ strm (CDR strm))
\end{verbatim}

are both true.

TEREAD — function + compiler macro

(TEREAD [strm])

This operator forces an end of line condition in the fast stream `strm`. Any characters left in the current input buffer are lost. A second application of TEREAD with no intervening NEXTs will have no effect.

If `strm` is omitted it defaults to CURINSTREAM.

TEREAD does not refresh the buffer in a stream, it simply puts the stream into the end-of-line condition, so that the next application of NEXT will refresh the buffer.

PUTBACK — function
(PUTBACK item strm)

One of the operators for manipulating input streams. The value of item is pushed onto the head of the stream, where it becomes the current object at the head of the stream.

item is always a character object when PUTBACK is called by the system.

This operation is, approximately, an output of one character to an input stream.

It is meaningful only for input streams, and finds application in the READ programs, where a delimiter is encountered by an auxiliary function, using ITEM-N-ADV, and the delimiter is PUTBACK onto the stream where it will be subsequently seen by that part of the READ operation which is equipped to interpret it.

It is needed to allow look ahead on streams which are really record oriented. In fact, given YKTLISP's definition of streams, PUTBACK is only required for look ahead greater than one if NEXT is used rather than ITEM-N-ADV. This implies, however, that end-of-lines must be explicitly handled.

OUTPUT

WRITE — function + compiler macro

(WRITE character strm)

This operator places character into strm

If the buffer is full it is written onto the output device, and character becomes the first character of the new buffer. (See "Streams" on page 33)

WRITE does not check strm to verify that it is an output stream.

PRINTCH — function + compiler macro

(PRINTCH character [strm])

This operation is equivalent to WRITE.

If strm is omitted it defaults to CUROUTSTREAM.

TERPRI — function + compiler macro

(TERPRI [strm])

This operator forces output of the current line in STREAM. A second application of TERPRI, with no intervening WRITEs, will result in a blank line.

If strm is omitted it defaults to CUROUTSTREAM.

WRITE-LINE — function

(WRITE-LINE c-str strm)

PRINTEXP — function + compiler macro

Writes c-str onto strm, as a single line. Only the contents of C-str are written, with no string delimiters or letterizer characters.

Any data written to the stream since the previous TERPRI (explicit or implicit) is lost. (PRINTEXP C-str [strm])
This operator writes the characters comprising C-str into STRM, with no string delimiters, and no letterizer characters.

If STRM is omitted it defaults to CUROUTSTREAM.

**SKIP** — function + compiler macro

(SKIP S-int [STRM])

This operator issues S-int TERPRI's to STRM. The value of SKIP is NIL.

If STRM is omitted it defaults to CUROUTSTREAM.

**TAB** — function + compiler macro

(TAB S-int [STRM])

This operator causes sufficient blanks to be written into STRM so that the last character in the stream output buffer is a byte position S-int. If there are already more than S-int bytes in the current output buffer, a TERPRI is performed, and S-int blanks inserted into an empty output buffer.

The effect of (TAB n STRM) is to cause the next character written to STRM to be placed in position n+1 in its buffer.

If STRM is omitted it defaults to CUROUTSTREAM.

**PRIN1** — function + compiler macro

(PRIN1 ITEM [STRM])

This operator writes the characters making up the representation of ITEM to STRM. No formatting is performed and no TERPRI is done. No blank is printed after ITEM. Thus the next write to STRM (if no TERPRIs intervene) will place characters immediately following the results of this operation.

This operator will print only non-descendible objects (i.e., neither pairs nor reference vectors). The value of PRIN1 is ITEM.

If STRM is omitted it defaults to CUROUTSTREAM.

**PRIN1B** — function + compiler macro

(PRIN1B ITEM [STRM])

This operand is similar to PRIN1, but a blank character is written into STRM after ITEM is printed.

If STRM is omitted it defaults to CUROUTSTREAM.

**PRIN0** — function + compiler macro

(PRIN0 ITEM [STRM])

This operator is the standard YKTLISP print routine, which writes the canonical output representation of the value of ITEM to STRM. PRIN0 is defined for all data objects. This representation shows all shared substructure, including cyclical substructure.

No formatting is performed and no TERPRI is done. No blank is printed after ITEM. Thus the next write to STRM (if no TERPRIs intervene) will place characters immediately following the results of this operation.

PRIN0 differs from PRIN1 in its ability to print pairs and reference vectors.

If STRM is omitted it defaults to CUROUTSTREAM.

**PRINT** — function + compiler macro
(PRINT item [strm])

This operator is equivalent to an application of PRIN0 followed by a TERPRI. Thus, the next write to strm will start a new line. Its value is item.

If strm is omitted it defaults to CUROUTSTREAM.

PRINM — function + compiler macro
(PRINM item [strm])

A specialized print operator which expects item to be a pair (otherwise item is CONSed with NIL and this pair is treated as item) and prints each element of the list item, with blanks between the elements. There are no top level parentheses printed, and if any element of the list item is a character string, it is printed by PRINTEXP instead of the normal, PRIN0, routine, so that its delimiting characters are not printed.

If strm is omitted it defaults to CUROUTSTREAM.

PRETTYPRINT — function + compiler macro
(PRETTYPRINT item [strm])

Similar to PRINT, except a more complicated program is invoked which understands many of the more common forms of symbolic expressions and prints them in a structured format. Unlike PRINT, PRETTYPRINT will not try to print structures with cycles in them. If a cycle exists in item, the regular PRINT function is invoked. PRETTYPRINT does not attempt to evidence shared structures as does PRINT, but prints each shared substructure in full. If it is not provided, the current value of the variable CUROUTSTREAM is used. PRETTYPRINT uses a free variable, PRETTYWIDTH, to define for it the maximum length of output lines.

If strm is omitted it defaults to CUROUTSTREAM.

PRETTYPRIO — function + compiler macro
(PRETTYPRIO item [strm])

This is a subfunction of PRETTYPRINT which takes the same arguments but does not perform a TERPRI after item has been printed.

The following operator does not use a stream.

CONSOLEPRINT — function + compiler macro
(CONSOLEPRINT c-str)

This operator displays its operand directly on the console device. It is the operator of last resort, to be used if you suspect that the world is coming down about your ears.

ACCESSING COMPONENTS

All streams are pairs. A simple list may be treated as a stream, however the majority of streams used in YKLISP have a complex structure, and are referred to as fast streams.

At the moment, fast streams are not a distinct data type, they are an interpretation of a particular structure of pairs and vectors. See "Streams" on page 33. YKLISP provides operators for accessing and updating the components of streams. These should be used in preference to CAR and ELT, as they will give protection against any change in the format of streams.

CAR — function
(CAR strm)

The value of CAR, when applied to a stream, is the current character. That is, for an output stream, the last character written into the stream, for an input stream, the last character extracted from its buffer.

CAR of a stream which is at end-of-line is EQ to the stream itself.

STREAM-A-LIST — function

(STREAM-A-LIST strm)

Returns the a-list of strm. This corresponds to the (possibly augmented) first operand of the DEFISTREAM invocation which originally created strm. See page 117.

STREAM-BUFFER — function

(STREAM-BUFFER strm)

Returns the I/O buffer of strm. This is normally a string, containing the last line read (for input streams), or the next line to be written (for output streams).

The TERPRI operator empties the buffer, using CHANGELENGTH to transform it into an empty string. A stream which has not been activated will have a buffer of NIL. Once a WRITE or NEXT has been performed on the stream, the buffer will be created, with the proper length.

STREAM-_DESCRIPTOR — function

(STREAM-Descriptor strm)

Returns the descriptor from strm. This is a reference vector (see "Streams" on page 33) containing various sub-components of the stream.

STREAM-P-LIST — function

(STREAM-P-LIST strm)

Returns the system dependent control block from strm. This is an object (a string or word vector) which is created, updated and used by the system dependent routines. It contains arbitrary data, not amenable to manipulation by LISP operators. In an unactivated stream the p-list will be NIL.

UPDATING COMPONENTS.

SET-STREAM-A-LIST — function

(SET-STREAM-A-LIST strm list)

Replaces the current a-list of strm by list.

SET-STREAM-BUFFER — function

(SET-STREAM-BUFFER strm item)

Replaces the current buffer in strm by item.

SET-STREAM-P-LIST — function

(SET-STREAM-P-LIST strm item)

Replaces the current p-list of strm by item.
This section describes the operators used to manipulate specialized DASD files referred to as libraries or LISPLIBs. These files are similar to OS partitioned data sets. They contain members, which are representations of LISP data objects, in a form which can be loaded quickly (though writing them is often slow).

Each member in a library is addressed by means of a key, a string, and has a class number (between 0 and 255) associated with it.

When a library is opened (by RDEFIOSTREAM) its directory is read and made part of the stream. If the library is opened for output, the file is so marked, and may not be opened for input until it has been shut. Output operations only modify the stream's directory, not that in the file, and the data is appended to the end of the file, not overwritten upon old data. Only when the stream is closed is the directory in the file updated.

Some of the following operators act directly on named libraries, while others act indirectly, through streams.

**CREATION**

RDEFIOSTREAM — function

(RDEFIOSTREAM list)

This operator creates a stream which can be used to access or update a library.

list has the same general form and meaning as the corresponding operand of DEFIOSTREAM, page 117, however the only meaningful properties are MODE and FILE.

**INPUT**

RREAD — function

(RREAD c-str rstrm)

This function reads the value component of a member, identified by c-str from the library specified by rstrm. c-str must be a character string. rstrm must have have been defined by RDEFIOSTREAM. If output has been performed to the library after the creation of rstrm, its effects will not be seen by RREAD. Such changes are only evident in streams which were created after the writing process has shut the library. This is so because the directory in the file is only updated upon shutting.

The value of RREAD is the object read.

An error break is taken if c-str is not a valid key for the library.

RCCLASS — function

(RCLASS c-str rstrm)

This operator returns the class value for the item designated by c-str in the library accessed by rstrm. The value is a small integer, in the range 0 to 255.

An error break is taken if c-str is not a valid key for the library.

RKEYIDS — function
(AKEYIDS file)

This operator returns a list of identifiers corresponding to thekeys in the library named by file. These identifiers are produced by INTERMing the actual keys, that is the identifier ABC represents the key 'ABC'.

OUTPUT

RWRITE — function

(RWRITE c-str item rstrm)

This operator add, or replaces, an object representing item associated with the key c-str to the library accessed by rstrm

If there already exists such an object the class is unchanged, if this is a new key, the class is set to zero.

Neither state descriptors nor bpis can be written directly to a library. State descriptors have no representation in libraries. Bpis are represented, but the representation must be created directly by the LISP assembler. This is accomplished by appropriate use of the FILE option in the definition option list.

RSETCLASS — function

(RSETCLASS c-str s-int rstrm)

This operator sets the class component for a member of a library. s-int must be in the range 0 to 255.

The operation is only valid if the key already has been added to the library, by an RWRITE, and an error break is taken otherwise. The effect is only visible after the library is shut.

RSHUT — function

(RSHUT rstrm)

For input streams the RSHUT operator simply performs a CMS FINIS.

For output streams the RSHUT operator writes the current library directory into the file, removes the "open for output" flag and performs a CMS FINIS.

A library which has been opened for output must be shut before it can be read again. Failure to do so will result in an unreadable file.

LIBRARY MANAGEMENT

RPACKFILE — function

(RPACKFILE file)

This operator reformats a library, discarding all inaccessible data.

RWRITE, it should be remembered, never overwrites old data representations, but rather appends new data to the end of the existing library. RPACKFILE removes the superseded data, shrinking the library to its minimal size.

If any errors occur during the processing, the original file is left unharmed.
RCOPYITEMS — function

(RCOPYITEMS file1 file2 list)

This operator copies selected members from file1 to file2. list is expected to be composed of identifiers, the pnames of which are used as keys to select members from file1. These members are written into file2, replacing existing members with the same keys. Any element of list which does not correspond to an existing member of file1 is ignored.

If errors occur during the processing file2 is unchanged.

RDROPITEMS — function

(RDROPITEMS file list)

This operator removes selected members from the library named by file. list is interpreted in the same manner as in RCOPYITEMS, and designates the members to be dropped from the library.

Only the directory is changed, to recover the DASD space occupied by the data the operator RPACKFILE must be used.

LIBRARIES AS TXTLIBS

This set of operators is used to load collections of operator definitions from libraries. They also contain facilities for "load time" evaluation of arbitrary expressions, allowing such actions as the setting of property values.

All of these operators are controlled by the class value of the members of a library. At the moment only the values 0 (zero) and 1 (one) are used.

A member with class 0 will be assigned as the value of the identifier corresponding to the key of that member.

A member with class 1 will be passed to the interpreter for evaluation.

LOADVOL — function

(LOADVOL file)

This operator reads all the members of the library named by file, and based on their class, assigns or evaluates them.

The value of the operator is a list of the identifiers corresponding to the keys in the library. This is equivalent to the value of RKEYIDS when applied to the same library.

SUBLOAD — function

(SUBLOAD file-name (id | list))

This operator attempts to read the members with keys corresponding to id or the elements of list from the library named by file-name. Each such existing member is treated as in LOADVOL.

The value is a two element list, (11 12), where 11 is a list of all elements of list (or id) which were found in file-name, and 12 is a list of all those not so found.

LOADCOND — function

(LOADCOND file)

This operator operates like LOADVOL with two differences. First, only members with class 0 are loaded, no load-time evalu-
actions are done. Second, the identifiers corresponding to the keys in the library are evaluated and any which currently have operators as values, (LAMBDA expressions or bpi's), are ignored. All others are loaded and assigned.

This allows various libraries to be constructed for different applications, with overlapping sets of operators. Suppose two such libraries exist, for example a cross reference analyzer for symbolic code and for compiled code. There could be certain operators needed by both packages. These operators can be replicated in the two libraries. Then, by using LOADCOND, both packages can be loaded without loading multiple copies of the common operators.

The restriction to class 0 members is required, as no simple test can be made to determine if a class 1 member has already been evaluated.

The value is a list of the identifiers corresponding to the members loaded.

**DEPLOY** — function

(DEPLOY (file-name | list1) (id | list2))

The first operand of DEPLOY is either a library name or a list of library names. The second is either an identifier or a list of identifiers.

All of the libraries named in the first operand are opened and searched for members corresponding to the identifiers in the second argument. All those found are loaded.

Then, every such loaded member is examined. If it is a bpi, the list of identifiers which it uses as operators is extracted from it. These are evaluated, and if their value is not an operator (LAMBDA expression or bpi) the libraries are searched for a member by that name. If one is found it is loaded, and its operator usage is similarly checked.

The value is a list, (ll 12), where ll is a list of identifiers corresponding to the members loaded, which 12 is a list of operator names which did not have operator values and which could not be found in the set of libraries.
In YKTLISP an identifier may be established as an operator by simple assignment. For various reasons this is rarely done in practice. Instead one of a group of definition operators is usually used.

This operators perform more or less preprocessing on the operator definition before making the assignment. The preprocessing treats certain expressions specially, in particular the LAM construct (see page undefined). It may involve compilation of LISP expression into bits, the assembly of LAM (Lisp Assembler Program) code, which is not applicable per se, and the creation of LISPFILES files, which are analogous to TXTLIBs.

All of the definition operators interact with a data structure, the option list. This is an a-list which specifies various types of processing. The option list may be augmented by the user in various ways, as will be noted below.

**DEFINITION**

**DEFINATE — function**

```
(DEFINATE list1 id1 id2 list2)
```

Where `list1` has a value of either

```
(name expression)
```

or

```
((name expression) ...)
```

The values of the second and third operands are key words, indicating the form of the first operand and the desired processing, respectively. These are the specification and the action.

The fourth operand is a list of name/value pairs which is to be temporarily added to the head of the option list.

There are five specifications and six actions, and they are interrelated. For example, COMPILATION specifies that the input is in the same form that would have been produced by the action COMPILE. The actions are ordered, each requiring the result of the previous one. DEFINATE sequentially performs them in order, starting with the action which will accept the first operand as specified, and continuing until the requested action has been reached.

The final disposition of the value is controlled by the option list.

The specification describes the expression parts of the first operand.

**EXPRESS**

**EXPRESSION**

An EXPRESSION is any LISP expression. As an action, EXPRESS is the identity operator, no processing is done.

**DEFINE**

**DEFINITION**

The action, DEFINE, examines an EXPRESSION for the explicit operator LAM. If it finds one it performs the macro expansion of the LAM into a MLAMBDAS expression and transforms that into two name expression lists, with the second having a name of the form LAM:name.

This prevents the body of the LAM expression from being included at each instance of its use. It also causes...
the body of the LAM expression to be not lexically present, and thus to behave in the same way as LAMBDA expressions with regard to variable evaluation.

REALIZE REALIZATION

The action, REALIZE, examines the operator of the DEFINITION. If it has a macro as its value in the compilation environment (see "The environment of compilation." on page 42) an expansion is performed and the test is repeated.

Once the form is not a macro it is further examined. If its operator is explicitly LAMBDA or MLAMBDA no further processing is done. Otherwise it is transformed into

\[( (LAMBDA()) expression) \]

unless it is already in that form.

It will be noted that applying this form is equivalent to applying expression itself. The reason of the transformation is to provide a form which the compiler can process.

REDUCE REDUCTION

The action, REDUCE, replaces all sub-expression in the REALIZATION by their macro expansions. This is done top down.

As in REALIZE, the value of the operators in the compilation environment determines whether a macro expansion is done.

COMPILE COMPILATION

The action, COMPILATE, translates the REDUCTION to LAP code.

LAP is a high level assembler, accepting S/370 machine instructions (in parenthesized form) and various pseudo-operations. It is not interpretable.

ASSEMBLE

The action, ASSEMBLE, translates LAP code into object code, and produces either a bpi, or an entry in a LISPLIB (or both).

There is no specification, ASSEMBLY, as there is no READable representation for the result of this action.

All other definition operators invoke DEFINATE with differing actions requested.

Their first operand is of the same form as the first operand of DEFINATE, their second as the last. They all specify their first operand as an EXPRESSION.

In actual usage these operators are almost always QUOTEd.

\[
(DEFINE "(READ (LAMBDA (X) (CAR X)))
 (TAII (LAMBDA (X) (CDR X))))\]

Note that the final result, whether a bpi, an addition to a LISPLIB or both, is controlled by the value of the option list.

DEFINE — function

130 YKTLISP Program Description and Operations Manual
(DEFINE list1 [list2])

Where the operands are as previously described.
DEFINITE invokes DEFINITE with a requested action of REALIZE.
The results are disposed of according to the value of the option list.

COMPILE — function

(COMPILE list1 [list2])

Where the operands are as previously described.
COMPILE invokes DEFINITE with a requested action of ASSEMBLE.
The results are disposed of according to the value of the option list.
The result of DEFINITE with an action of COMPILE is not usable as an operator. The COMPILE operator translates LISP expressions to bpsi.

ASSEMBLE — function

(ASSEMBLE list1 [list2])

Where the operands are as previously described.
ASSEMBLE invokes DEFINITE with a requested action of ASSEMBLE and an operand specification of COMPILATION.
The results are disposed of according to the value of the option list.
ASSEMBLE is used to transform programs written in LAP into bpsi.

OPTION LIST

The values of various properties on this augmented list control various aspects of definition, compilation and assembly. GET is used to search OPTIONLIST, thus NIL is the default value for any property not explicitly present.

Various processes effect the option list. Every definition operator may temporarily add its own options. EXF add options which are removed when it finishes. Other operators change the current value.

Both EXF and DEFINITE re-bind OPTIONLIST, and the standard operators modify it in such a way that their effects are lost upon exit from these functions.

ADDOPTIONS — function

(ADDOPTIONS [id item] ...)

This operator constructs an s-list from its operands, using them alternately as names and values. The result is appended onto the front of the current value of OPTIONLIST.

If the number of operands is odd, an error break is taken.

ORADDTEMPDEFS — function

(ORADDTEMPDEFS file)

file must designate a LISPLIB. The objects read from file are added to the OP-RECOGNITION-SD, as the values of their keys.
This allows a set of macro definitions to be installed during the processing of a set of compilations, by EXF for example, without their becoming a permanent part of the system.

**MAADDTEMPDEFS — function**

```lisp
(MAADDTEMPDEFS file)
```

The `file` must designate a LISPLIB. The objects read from file are added to the MACRO-APP-SD, as the values of their keys.

This allows a set of functions needed by compiler macros to be available, without their interfering with the standard system functions.

**ORTEMPDEFINE — function**

```lisp
(ORTEMPDEFINE list1 [list2])
```

Where the operands are as for `DEFINE`.

`ORTEMPDEFINE` invokes `DEFIMATE` with a requested action of `REALIZE`.

The results are added to `OP-RECOGNITION-SD`. This allows temporary macro definitions, which will persist only as long as the current binding of `OPTIONLIST`, without interfering with the standard system definitions.

**MATEMPDEFINE — function**

```lisp
(MATEMPDEFINE list1 [list2])
```

Where the operands are as for `DEFINE`.

`MATEMPDEFINE` invokes `DEFIMATE` with a requested action of `REALIZE`.

The results are added to `MACRO-APP-SD`. This allows the temporary definition of functions needed by macros, without their interfering with the normal system definitions. These will persist only as long as the current binding of `OPTIONLIST`.

**OP-RECOGNITION-SD — key word**

```lisp
(OP-RECOGNITION-SD . sd)
```

The environment in which the operators are evaluated by the compiler. The values found in this environment are used to distinguish functions, macros and special forms. Various operators exist to augment this environment; see "The environment of compilation" on page 42.

**MACRO-APP-SD — key word**

```lisp
(MACRO-APP-SD . sd)
```

The environment in which macros are to be expanded by the compiler. The use of `MACRO-APP-SD` protects from conflict between variables bound by the compiler and the bindings of macros used in expressions being compiled. If the value is `NIL`, the initial state (where only nil-environment bindings are present) is used.

**NOLINK — key word**

```lisp
(NOLINK . boolean)
```

If the `NOLINK` property has a non-`NIL` value the result of an assembly is not made into a bpi. The default is to create a bpi and assign it to the name with which it was paired in the specification list.

**INITSYMTAB — key word**
(INITSYM TAB . list.)

The value of this property should be an association list (or NIL, which is an empty association list) which will be searched by the assembler (LAP) for operation code values, symbolic register names, symbolic immediate operand names and symbolic literals. The values in INITSYM TAB override the built-in values of the assembler, and are overridden in turn by symbols established by ERU statements in the LAP code.

NONINTERRUPTIBLE — key word

(NONINTERRUPTIBLE . boolean)

If the NONINTERRUPTIBLE property is non-NIL, no polling for interrupts is inserted in the bpi by the assembler. Explicit POLL statements will be assembled. The default value is NIL, i.e. the bpi is interruptible.

SOURCE LIST — key word

(SOURCELIST . boolean)

If SOURCELIST has a non-NIL value the source program (either LISP or LAP) is PRETTYPRINTed by the definition functions. The default is NIL. When running with SOURCELIST non-NIL it should be remembered that the printing by the supervisor can be controlled by the settings of the fluid variables 'ECHO$W and/or ',VALUS$.

TRAN SLIST — key word

(TRANSLIST . boolean)

A non-NIL value for TRANSLIST causes the output of pass one of the compiler to be PRETTYPRINTed. This is the "transformed" LISP, with all macros expanded and with various other changes, which will be made into a bpi by pass two of the compiler and the assembler. A number of forms internal to the compiler appear in this listing. If definitions of these internal forms existed (unfortunately impossible in some cases) interpretation of this transformed LISP would duplicate the behavior of the bpi which results from the full compilation/assembly process. The default value is NIL.

LAP LIST — key word

(LAPLIST . boolean)

A non-NIL value for LAPLIST causes the assembly code produced by the compiler to be PRETTYPRINTed. This property does not control the printing of LAP source code, which is under the control of the SOURCELIST property. The default value is NIL.

BPI LIST — key word

(BPILIST . boolean)

A non-NIL value for BPILIST causes an assembly listing to be produced. This listing contains the hexadecimal System/370 machine code produced by the assembler, together with a variable amount of symbolic LAP code.

If BPILIST is non-numeric or if it is greater than 3 a full listing is produced. This includes all instructions generated by the assembler, all comments and all source instructions.

A value of 3 causes intermediate instructions to be dropped from the listing. That is, instructions generated by the assembler, which in turn resulted in the generation of further instructions rather than in object code.

A value of 2 causes comments to be dropped from the listing.
A value of 1 causes only source instructions to be printed symbolically, although the object code (hexadecimal) is printed in full.

**LISTING** — key word

(listing . strm)

The value of LISTING should be NIL or a fast stream. If LISTING is a stream the output produced as a result of the preceding four options (SOURCELIST, TRANSLIST, LAPLIST and BPILIST) will be written onto that stream. If LISTING is NIL it defaults to the value of the fluid CUROUTSTREAM.

**MESSAGE** — key word

(message . strm)

The value of MESSAGE should be NIL or a fast stream. All error and warning messages from the definition functions are written onto the MESSAGE stream. If the MESSAGE and LISTING streams are not EQ the MESSAGE stream is made to dominate the LISTING stream. (See the DOMINATESSTREAM function.) If the value of MESSAGE is NIL it defaults to CUROUTSTREAM.

**FILE** — key word

(file . rstrm)

The value of FILE should be NIL or a stream. If FILE is non-NIL a loadable bpi-image will be written onto it. If FILE is NIL no action is taken.

By use of the FILE and NOLINK options programs can be compiled and/or assembled for future loading without their being defined in the running system. (See LOADVOL function.) The resulting file, when loaded, causes the same assignments and/or MAKEPROPS to take place as would have resulted if the NOLINK option were NIL. The current form of a loadable file may be changed in the future, however this is still a matter of dispute and discussion.

**QUIET** — key word

(quiet . boolean)

If the QUIET property has a non-NIL value warning and informational messages to the console are suppressed.

**OPTIMIZE** — key word

(optimize . s-int)

The value of the OPTIMIZE property control the amount of code modification during pass II of the compiler. The default value is currently 4, the present highest level of optimization. Level 0 lets the code emitted by the various generators stand as-is. The levels from 1 to 3 perform various operations, such as dead code elimination, code motion to eliminate branches, common code merging, etc. There is no guarantee that 4 will remain the top level (and hence the default). If time permits further optimizations may be added.
ENVIRONMENT EXAMINATION

& — macro

(& [id [item ...]])

The & operator controls the examination of stack frames created by the execution process. It is possible to look at arguments of functions, to determine bindings of variables, and to display variable values.

The first argument, if present, specifies a command to the operator, the remaining arguments may be in arbitrary order.

id The COMMAND, an identifier or list specifying the action, i.e., the kind of stack examination desired. The possibilities are:

INDEX Display a series of stack frame identifications sequentially indexed with a small integer. These appear in a LIFO order (last in execution, first in listing). The form of each frame identification is:

1. index, or frame number
2. frame name, or NIL
3. frame type
4. contour level, if not outermost.

FULL Stack frame identification followed by

1. argument names, if any arg
2. all variables, with their been bound at this frame.

(List of identifiers) For every identifier there is an indication of lexical or fluid binding, stack frame identification, and value.

If the identifier refers to a generated symbol, i.e., one in the form %n, only the numeric part, i.e., n, should appear in the list.

FLUID Stack frame identification for only those frames at which fluid binding information exists; following each identification are all the variables, and their values, with FLUID bindings at that frame.

LEX Stack frame identification for only those frames at which lexical binding information exists; following each identification are all the variables, and their values, bound lexically at that frame.

UNWIND Display information for each stack frame which has a catch point. Those which are known to the & operator are annotated. See UNWIND, page 57, and Figure 14 on page 56.

BIND Stack frame identification for only those frames at which some binding information exists; following each identification are all the variables bound at that frame.

ARGS Stack frame identification for only those frames representing functions to which
arguments have been passed; following each identification are all the argument names and their values.

**SECD** Stack frame identification for only those frames associated with SECD (interpretive) execution and which also have some elements on the SECD control or stack; there follows the elements of the control and stack.

The remaining arguments may be given in arbitrary order. The only caveat is with respect to the START and STOP points, where the relative order of appropriate values (N, numbers) defines their meaning.

**chain** The search control.

**A** Indicates that the environment chain is to be examined. If not present, the control chain is examined.

**print** The print control. The default is an elided display, one line per variable.

**PP** Indicates that PRETTYPRINTing is to be done. This option is useful when a variable must be examined in detail.

**PR** Indicates that the normal PRINT function is to be used. This will show it existence of shared sub-structure.

(PR operator) Indicates the the supplied operator is to be used for printing.

**floor** Reference frame specification.

The frame numbers displayed by the & operator and used in the start and stop operands are relative to a "floor" frame. Normally this is the nearest frame on the control chain which is in use by the operator EVALFUN. This choice prevents the internal workings of the & operator from distracting the user.

It is possible to specify another frame as the "floor" frame.

(FL s-int) use the frame s-int above the one being used by the display function as the floor. The display function itself is frame 0.

(FL id) where id has a bpi as its value. Use the nearest frame (in the chain specified by Chain) which is in use by the bpi which is the value of id.

**start** Start control.

s-int1 Specifies the first stack frame which is to be examined.

start defaults to 1.

**stop** Stop control.

s-int2 Specifies the last stack frame which is to be examined.

N Specifies that the stack examination should continue through to the highest level.

stop defaults to:

start + 15 when id is INDEX and start was specified.
10 when id is FULL and start was not specified.
* when id is UNWIND and any other case when start was specified.
start in any other case when start was specified.

value request for value
Must be used with the variable list command.

VAL Returns the value of the first variable found.
VALLIST Returns a list of all values of all variables found.

(\), with no operands is equivalent to (\ index 1 14).

We will compile a version of the factorial function which forces an error break when its argument is zero.

```
(COMPILE "(FACT
  (LAMBDA (N)
    (COND ((EQ N 0) (BREAK)) (\T (TIMES N (FACT (SUB1 N)))))))
)
```

Macro-expanding FACT
Compiling FACT
Assembling FACT
Linking FACT
Value = (FACT)

We now call the new function.

```
(FACT 5)
Error: forced break,
Break taken,
FIN with any value, otherwise UNWIND to exit break loop.
(\)
\ARGS? = NIL
1 %SUBR.S.ERRORLOOP = Contour: 1
2 %SUBR.S.ERRORLOOP
3 %SUBR.CONDERR
4 %SUBR.BREAK
5 %SUBR.FACT
6 %SUBR.FACT
7 %SUBR.FACT
8 %SUBR.FACT
9 %SUBR.FACT
10 %SUBR.FACT
11 SECD1
12 %SUBR.EVA1FUN
13 %SUBR.SUPV = Contour: 1
14 %SUBR.SUPY
NIL
```

Note that (\) displays the value of the variable \ARGS?, which is sometimes used to pass information about the error (see below).

We now ask for more information about four of the stack frames.
We request a complete back-trace of the stack.

```
(& index)
1 % .SUBR.S,ERRORLOOP = Contour: 1
2 % .SUBR.S,ERRORLOOP
3 % .SUBR.CONDERR
4 % .SUBR.BREAK
5 % .SUBR.FACT
6 % .SUBR.FACT
7 % .SUBR.FACT
8 % .SUBR.FACT
9 % .SUBR.FACT
10 % .SUBR.FACT
11 SECDI
12 % .SUBR.EVAIFUN
13 % .SUBR.SUPV = Contour: 1
14 % .SUBR.SUPV
15 % .SUBR.SUPERMAN = Contour: 2
16 % .SUBR.SUPERMAN = Contour: 1
17 % .SUBR.SUPERMAN
18 % .SUBR.HIGHLORD
NIL
```

We request information about all bindings of N.

```
(& (N))
N
  Lexical 5 % .SUBR.FACT
  0
  Lexical 6 % .SUBR.FACT
  1
  Lexical 7 % .SUBR.FACT
  2
  Lexical 8 % .SUBR.FACT
  3
  Lexical 9 % .SUBR.FACT
  4
  Lexical 10 % .SUBR.FACT
  5
NIL
```

We request information about catch-points in the control chain.

138  YKTLISP Program Description and Operations Manual
We request information about an interpreter frame.

We leave the break loop, providing a value for the call to BREAK.

Now we will force some system-detected errors.

Here, ?ARGS? holds the operator (CAR) and the invalid operand (NIL).

We will UNWIND to the supervisor and force another error.
(UNWIND 1)

Unwind caught by SUPV with value: NIL

(FOO X)

Error: application of inapplicable object,
FIN with any value, otherwise UNWIND to exit break loop.

(ARGS? = (X FOO)

(LAST ?ARGS?) = FOO

1 %.SUBR.S.ERRORLOOP = Contour: 1
2 %.SUBR.S.ERRORLOOP
3 %.SUBR.CONDERR
4 SECID
5 SECID1
6 %.SUBR.EVA1FUN
7 %.SUBR.SUPV = Contour: 1
8 %.SUBR.SUPV
9 %.SUBR.SUPERMAN = Contour: 2
10 %.SUBR.SUPERMAN = Contour: 1
11 %.SUBR.SUPERMAN
12 %.SUBR.HIGHLORD

NIL

Here, again, ?ARGS? shows us the operator which is in error.

CALL TRACING

TRACE macro

(TRACE exp id ...)

exp is evaluated, with all calls to the operators which are the values of id ... traced.

(TRACE
  (PLUS (TIMES 12 20) (QUOTIENT 13 4.3))
  PLUS
  TIMES
  QUOTIENT)
TIMES Called by EVA1FUN
  Args = (12 20)
  Value = 240
TIMES Returned
QUOTIENT Called by EVA1FUN
  Args = (13 4.3)
  Value = 3.023255813953
QUOTIENT Returned
PLUS Called by EVA1FUN
  Args = (240 3.023255813953)
  Value = 243.023255814
PLUS Returned
Value = 243.023255814

140  YKTLISP Program Description and Operations Manual
MONITOR — function

(MONITOR id [list1 [list2]])

MONITOR is a simple call tracing operator. The first argument is the id, the value of which is the function or macro to be traced. Once MONITOR has been executed, all calls to id will be intercepted and the values of the arguments, followed by the value of the call to id, or the expansion of id if it is a macro, will be printed on CUROUTSTREAM. If id is MONITORed in compiled code, the id of the calling program will also be printed.

The two optional arguments are lists of IDs. If they are present and non-NIL the FLUID bindings of the IDs in the list1 will be printed before the MONITORed function is actually called, while those in the list2 will be printed after the function returns.

The effect of MONITOR is removed by (UKEMBED id).
Macro-expanding STEP-X

Compiled STEP-X

Assembling STEP-X

Linking STEP-X

Value = (STEP-X)
(MONITOR "STEP-X "(X) "(X))
Value = STEP-X
(STEP-X 4)

X = X
STEP-X Called by EVALFUN

Args = (4)
Value = 4

STEP-X Returned

X = 4
Value = 4
(STEP-X 4)

X = 4
STEP-X Called by EVALFUN

Args = (4)
Value = 8

STEP-X Returned

X = 8
Value = 8
(STEP-X 4)

X = 8
STEP-X Called by EVALFUN

Args = (4)
Value = 12

STEP-X Returned

X = 12
Value = 12

(MONITOR "PROG"

Value = PROG
(PROG (A B)

(SETQ A 1)

(SETQ B 2)

(SETQ A (CONS A B))

(RETURN A))

PROG Being MACRO-expanded

Args = (PROG (A B)

(SETQ A 1)

(SETQ B 2)

(SETQ A (CONS A B))

(RETURN A))

Expansion = ((LAMBDA (A B)

(SEQ

(SETQ A 1)

(SETQ B 2)

(SETQ A (CONS A B))

(RETURN A))

NIL

NIL)

PROG Returned

Value = (1 2)
(UNEMBED "PROG"

Value = PROG

UNEMBED — function

(UNEMBED id)

Removes the effect of an EMBED of the value of id. In partic-
ular, cancels the effect of a use of MONITOR on id.

142 YKTLISP Program Description and Operations Manual
EMBED — function

(EMBED id exp)

Where the value of id is normally an operator, exp', and exp is a LAMBDA or MLAMBDA expression.

If exp does not contain instances of id, then EMBED is effectively a reversible assignment of exp to id.

If exp contains instances of id, then exp is modified in such a way as to cause those instances to evaluate to exp', the previous value of id. All other uses of id (exterior to exp) will evaluate to this modified expression.

(DO

(define (FACT (LAMBDA (N))
  (cond
   ((EQ N 0) 1)
    ((T (TIMES N (FACT (SUB1 N)))))))

Value = (FACT)
(FACT 6)
Value = 720
(EMBED
 "FACT
 "((LAMBDA (X)) (PRINT 'Hi there!') (FACT X))))

Value = FACT
(FACT 6)
'Hi there!'
'Hi there!'
'Hi there!'
'Hi there!'
'Hi there!'
'Hi there!'
Value = 720

EMBEDDED — function

(EMBEDDED)

Returns a list of all ids which currently have EMBEDed definitions.
### Table of System Functions, Variables and Commands

See Figure 13 on page 44 for a description of the operand types.

- **.NOVAL**
  - **variable**
  - Value is the "undefined" object.

- **&**
  - **macro**
  - `(i id [item ...])`
  - back trace printer, unquoted arguments. See page 135.

- **$ALLFILES**
  - **CMS dependent function**
  - `($ALLFILES file-name)`
  - Returns a list of all files that match the arguments.

- **$CHECKMODE**
  - **CMS dependent function**
  - `($CHECKMODE item)`
  - Returns () if a CMS filemode for a linked disk.

- **$CLEAR**
  - **CMS dependent function**
  - `(CLEAR)`
  - Clear the screen if the console is a 32xx.

- **$ERASE**
  - **CMS dependent function**
  - `($ERASE file-name)`
  - Erase a file.

- **$EST**
  - **CMS dependent function**
  - `($EST [str | id])`
  - Tests the status of the disk with the given filemode.

- **$FCOPY**
  - **CMS dependent function**
  - `($FCOPY [str | id] [str2 | id2] id3)`
  - Copy a file. If id3=APPEND then append.

- **$FILEDATE**
  - **CMS dependent function**
  - `($FILEDATE [str | id])`
  - Return a string of the form 'yy/mm/dd hh:mm' or ().

- **$FILEREC**
  - **CMS dependent function**
  - `($FILEREC [str | id | s-int] s-int)`
  - Returns a string containing the specified record or ().
  - Will read from files or console.

- **$FILESIZE**
  - **CMS dependent function**
  - `($FILESIZE [str | id])`
  - Returns () or the file size in bytes (this is a pessimistic upper bound for v-format files).

- **$FINDLINK**
  - **CMS dependent function**
  - `($FINDLINK [str | id])`
  - Return a modeletter if the argument denotes a linked disk.

- **$FLAT-STRING**
  - **function**
  - `($FLAT-STRING item)`
  - Concatenate all the atoms in the argument ignoring any list structure and converting atoms as required.

- **$FNFFTM**
  - **function**
  - `($FNFFTM file)`
  - Verify arguments for reasonableness and return a list (fn ft fm).

- **$INFILE**
  - **function**
  - `($INFILE file)`
  - Value is (fn ft fm) if the file exists, otherwise ().

- **$INSTREAM**
  - **function**
  - `($INSTREAM file-name s-int)`
  - Value is a stream positioned at the given record if the file exists, otherwise ().
$OUTFILE
function
($OUTFILE file)
Value is (fn ft fm) if it can be an output file.

$OUTSTREAM
function
($OUTSTREAM file-name s-int1 s-int2)
Value is an output stream positioned at the specified record if possible. The width argument is used only for a new file.

$READFLAG
function
($READFLAG)
Value is non-() if there are lines in the program (or console) stacks.

$REPLACE
function
($REPLACE file-name1 file-name2) The data in file-name2 replaces the data in file-name1. file-name2 is erase, and the data in file-name1 is lost.

$RESET-STREAM
function
($RESET-STREAM strm)
Update the stream so that the next lisp read operation will start at the first character of the current record.

$SCREENSIZE
function
($SCREENSIZE)
If the current console is a 32xx, the value is (rows cols) or (rows cols REMOTE).

$SETPFIMM
function
($SETPFIMM s-int c-str)
Set the PF key to the string as an immediate command.

$SHOWLINE
function
($SHOWLINE c-str s-int)
Display the string on the specified line of a 32xx.

$T-Delta
function
($T-Delta num)
Value is elapsed total cpu time.

$T-TIME
function
($T-TIME)
Total cpu time in milliseconds since last logon.

$TIMESTAMP
function
($TIMESTAMP)
Value is current date and time as a string 'mm/dd/yy hh:mm'

$TOKEN
function
($TOKEN c-str s-int)
Value is () or the index-th substring delimited by blanks.

$TYPELINE
function
($TYPELINE c-str)
Type the string at the console.

$V-Delta
function
($V-Delta num)
Value is virtual cpu time in milliseconds since num.

$V-TIME
function
($V-TIME)
Value is virtual cpu time in milliseconds since last logon.

%CODE
special form
(%CODE exp list [lap-statement ...])
Where exp is the interpreative equivalent and list is a declaration. Inserts in-line LAP code.
function
(\? command [item \ldots])
Synonym for LAM.\&

? command to system interface
(CALLBELOW '?' [c-str | s-int])
Query to SYSDEP, to verify commands.

? key word
Request to the break loop to repeat the error message.
See page 17.

ABSVAL
function
(ABSVAL num)
Absolute value. See page 107.

ADDOPTIONS
function
(ADDOPTIONS [id item] \ldots)
Adds pairs to OPTIONLIST See page 131.

ADDRESSOF
function, with macro definition for compilation
(ADDRESSOF item)
Returns the address part of pointer as SMINT

ADDTOLIST
function
(ADDTOLIST list item)
Adds an item to a list if not already there. See page 87.

ADD1
function
(ADD1 num)
Incrments its argument by one. See page 108.

ALINE
function
(ALINE $-int1$-int2)
Where $-int2$ is a power of 2. Rounds up to a power of two. See page 110.

ALL-NON-DESC
function
(ALL-NON-DESC item)
Value is a list of all the non-descendable components in the argument.

ALLFUNCTIONS
function
(ALLFUNCTIONS id | bpi)
Returns list of functions called by a BPI

AND
macro
(AND [exp \ldots])
Evaluates expressions until one returns NIL. See page 51.

ANDBIT
function
(ANDBIT b-str \ldots)
Logical AND of bit strings. See page 97.

APPEND
function
(APPEND item1 item2)
"Pastes" two lists, with some copying. See page 82.

APPLX
built in function
(APPLX app-ob list)
Apply a function to a list of values

APPLY
built in function
(APPLY app-ob list sd)
APPLX with respect to an environment

ARRAYKEYS
function
(ARRAYKEYS hasharray)
Value is a list of all the keys in the array.

ASMTIME
function
(ASMTIME)
Returns time and date of system assembly

Table of system functions, variables and commands 147
ASSEMBLE function
(ASSEMBLE list1 [list2])
Where list1 is either a name/LAP program list or a list
of name/LAP program lists.
The LAP assembler See page 131.

ASSOC function
(ASSOC item list)
Search a list of name-value pairs, uses EQUAL. See page
85.

ASSOCN function
(ASSOCN item list)
Search a list of name-value pairs, uses NEQUAL. See page
86.

ASSQ function
(ASSQ item list)
Search a list of name-value pairs, uses EQ. See page 86.

ATOM built-in function
(ATOM item)
Tests for not-pairness. See page 71.

AUGMENTGLOBAL function
(AUGMENTGLOBAL list sd)
Where list is of the form ((name . value) ...).
Add bindings to global environment of SD.

AUGMENTSTACK function
(AUGMENTSTACK list sd)
Where list is of the form ((name . value) ...).
Add FLUID bindings to stack environment of SD.

BATCHERROR function
(BATCHERROR item1 item2)
Action when break is entered in batch.

BITGREATERP function
(BITGREATERP b-str1 b-strn)
Compares two bitstrings. See page 103.

BITSTRINGP built-in function
(BITSTRINGP item)
Tests for bitstring type. See page 72.

BITSTRING2STRING function
(BITSTRING2STRING b-str)
Changes type and contents count of bit string. See page
96.

BITSUBSTRING2NUM function
(BITSUBSTRING2NUM b-str s-int1 s-int2)
Numeric value of up to 24 bits of bitstring.

BOOLEANP function
(BOOLEANP)
Synonym for TRUEP, to make nicer messages.

BOUNDEDBY? function
(BOUNDEDBY? bpi)
Searches environment chain for BPI in frame.

BOUND DP function
(BOUND DP id)
Tests for the existence of a binding.

BPILEFT function
(BPILEFT)
Returns bytes of BPI space remaining.

BPILIST definition option
(BPILIST . boolean)
Request printing of an assembly listing on the listing
stream. See page 133.
BPINAME  function  
(BPINAME bpi)  
Returns the "name" id of a BPI

BPIP  function  
(BPIP item)  
Tests for BPI data types. See page 73.

BREAK  function  
(BREAK)  
Forces an error break

BYTES  function  
(BYTES item)  
Value is a list, (bytes ids sds), that describes the storage occupied by the argument.

CNR  function, with macro definition for compilation  
(CNR C-str item)  
Where string has the form '{A | D}...'. Does nested CAR and CDRs

CAAAAR  function, with macro definition for compilation  
(CAAAAR item)  
(CAR (CAR (CAR x))). See page 77.

CAAADR  function, with macro definition for compilation  
(CAAADR item)  
(CAR (CAR (CAR (CDR x)))). See page 77.

CAAAR  function, with macro definition for compilation  
(CAAAR item)  
(CAR (CAR (CAR x))). See page 77.

CAADAR  function, with macro definition for compilation  
(CAADAR item)  
(CAR (CAR (CDR (CAR x)))). See page 77.

CAAADDR  function, with macro definition for compilation  
(CAAADDR item)  
(CAR (CAR (CDR (CDR x)))). See page 77.

CAADR  function, with macro definition for compilation  
(CAAADR item)  
(CAR (CAR (CDR x))). See page 77.

CAAR  function, with macro definition for compilation  
(CAAR item)  
(CAR (CAR x)). See page 77.

CADAAR  function, with macro definition for compilation  
(CADAAR item)  
(CAR (CDR (CAR (CAR x)))). See page 77.

CADADDR  function, with macro definition for compilation  
(CADADDR item)  
(CAR (CDR (CAR (CDR x)))). See page 77.

CADAR  function, with macro definition for compilation  
(CADAR item)  
(CAR (CDR (CAR x))). See page 77.

CADDAR  function, with macro definition for compilation  
(CADDAR item)  
(CAR (CDR (CDR (CAR x)))). See page 77.

CADDDDR  function, with macro definition for compilation  
(CADDDDR item)  
(CAR (CDR (CDR (CDR x)))). See page 77 and page 84.

CADDR  function, with macro definition for compilation  
(CADDR item)  
(CAR (CDR (CDR x))). See page 77 and page 84.

CADDR  function, with macro definition for compilation  
(CADDR item)  
(CAR (CDR (CDR x))). See page 77 and page 84.
CALL

built in function

(CALL (item ...) app-ob)

Apply last argument to list of other arguments

CALLBELOW

function, with macro definition for compilation

(CALLBELOW c-str (sysdep-area ...))

Invoke system dependent code

CALLEDBY?

function

(CALLEDBY? bpi)

Searches control chain for calling BPI

CALLX

built in function

(CALLX app-ob [item ...])

Apply first argument to remaining arguments

CAR

built in function

(CAR item)

First element of a pair. See page 77 and page 84.

CASEGO

macro

(CASEGO exp list ...)

Where list has the form (item id).

N way branch, uses EQ. See page 50.

CATCH

function

(CATCH id1 exp [id2 (item ...)])

Establishes stopping point for THROWs. See page 55.

CBOUNDP

function

(CBOUNDP id)

Test for binding on the control chain

CDAAAR

function, with macro definition for compilation

(CDAAAR item)

(CDR (CAR (CAR (CAR x))))). See page 77.

CDAADR

function, with macro definition for compilation

(CDAADR item)

(CDR (CAR (CAR (CDR x)))). See page 77.

CDAAR

function, with macro definition for compilation

(CDAAR item)

(CDR (CAR (CDR x))). See page 77.

CDADAR

function, with macro definition for compilation

(CDADAR item)

(CDR (CAR (CDR (CAR x))))). See page 77.

CDADDR

function, with macro definition for compilation

(CDADDR item)

(CDR (CAR (CDR (CDR x)))). See page 77.

CDADR

function, with macro definition for compilation

(CDADR item)

(CDR (CAR (CDR x))). See page 77.

CDAR

function, with macro definition for compilation

(CDAR item)

(CDR (CAR x)). See page 77.

CDDAAR

function, with macro definition for compilation

(CDDAAR item)

(CDR (CDR (CAR (CAR x))))). See page 77.

CDDADR

function, with macro definition for compilation

(CDDADR item)

(CDR (CDR (CAR (CDR x)))). See page 77.

CDDAR

function, with macro definition for compilation

(CDDAR item)

(CDR (CDR (CAR x))). See page 77.
CDDAR function, with macro definition for compilation
(CDDAR item)
(CDR (CDR (CDR (CAR x)))) See page 77.

CDDDR function, with macro definition for compilation
(CDDDR item)
(CDR (CDR (CDR (CDR x)))) See page 77 and page 84.

CDDR function, with macro definition for compilation
(CDDR item)
(CDR (CDR (CDR x))) See page 77 and page 84.

CDR built-in function
(CDR item)
Second element of pair. See page 77 and page 84.

CEVAL-ID function
(CEVAL-ID id)
Control chain ID evaluator, drops lexicals. See page 60.

CEVAL-LEX-ID function
(CEVAL-LEX-ID id)
Control chain ID evaluator, sees callers lexes. See page 61.

CHANGELENGTH function, with macro definition for compilation
(CHANGELENGTH str s-int)
Change the contents count of a string. See page 81.

CHARP function, with macro definition for compilation
(CHARP item)
Tests for one character identifier. See page 73.

CHAR2NUM function
(CHAR2NUM char)
Where identifier has a one character name.
Returns the number, 0 to 255, for character id

CLOSEDFN special form
(CLOSEDFN item)
Where item is a LAMBDA expression or an MLAMBDA expression. In interpreter = QUOTE, compiles to QUOTEd bpi. See page 54.

CLOSURE built-in function
(CLOSURE exp sd)
Creates a FUNARG from an expression and a SD

COMCELLP function
(COMCELLP item)
Test for existence of a communication cell

COMMENT macro
(COMMENT exp [item ...])
Expands to expr.

COMPILE function
(COMPILE list1 [list2])
Where list1 is either a name/expression list or a list of
name/expression lists.
The LISP compiler. See page 131.

COMP370 function
(COMP370 list1 [list2])
Where list1 is either a name/expression list or a list of
name/expression lists.
Old version of COMPIL e. hacks recursive macros

CONC function, with macro definition for compilation
(CONC item ...$rbk)
Multi-argument APPEND. See page 82.

Table of system functions, variables and commands 151
COND
special form
(COND [clause ...])
Where clause is either a string (a comment) or of the form (EXP ...).
Nested IF-THEN-ELSEs. See page 50.

CONDERR
function
(CONDERR exp item item2 app-ob)
Where item1 is either a string or a list of strings.
Enter to break, typed FIN or UNWIND enforced

CONS
built-in function
(CONS item1 item2)
Creates a new pair from its arguments. See page 77 and page 81.

CONSOLEPRINT
function, with macro definition for compilation
(CONSOLEPRINT C-str)
Output directly to console. See page 123.

CONSTANT
macro
(CONSTANT exp)
Where item will be evaluated at compile-time.
QUOTEs the value of its argument. See page 47.

CONTAINSQ
function
(CONTAINSQ item1 item2)
Searches item2 (any pair/vector structure) for EQ item1

CONVERSATIONAL
function
(CONVERSATIONAL)
Tests for batch/non-batch operation

CONVSAL
command to system interface
(CALLBELOW 'CONVSAL')
Tests for batch versus conv. mode

COPY
function
(COPY item)
Creates a structurally exact copy

COS
function
(COS num)
Cosine, argument in radians. See page 111.

COUNT
macro
(COUNT exp id ...)
Evaluate exp and count how many times id ... are called.

COUNT-CALLS
macro
(COUNT-CALLS exp)
Evaluate exp and count the total number of contours entered.

COUNT-PAIRS
macro
(COUNT-PAIRS exp)
Evaluate exp and count calls by caller and callee.

CREATE-SBC
function
(CREATE-SBC id)
Creates a shallow binding cell for an id

CSET-ID
function
(CSET-ID id item)
Control chain assignment, drops lexicals. See page 62.

CSET-LEX-ID
function
(CSET-LEX-ID id item)
Control chain assignment, sees callers lexes. See page 62.

CURINSTREAM
variable
Default input stream See page 117.

CURLINE
function

152 YKTLISP Program Description and Operations Manual
DIG2FIX  function, with macro definition for compilation
         (DIG2FIX digit)
         Convert identifiers |0 - |9 to small integers

DISABLE  function, with macro definition for compilation
         (DISABLE)
         Suppresses recognition of interrupts, cf ENABLE

DISPATCHER  function
         (DISPATCHER item s-int)
         System function, gets control on interrupt

DIVIDE  function, with macro definition for compilation
         (DIVIDE num1 num2)
         Quotient and remainder for two arguments. See page 109.

DROPAREA  command to system interface
         (CALLBELOW 'DROPAREA' s-int1 s-int2)
         Releases FREE/FRET storage.

EBCDIC  function
         (EBCDIC s-int)
         Returns the character id for s-int 0 to 255

ECONST  macro
         (ECONST s-int)
         Expands to quoted value of (EBCDIC s-int).

ECQ  macro
         (ECQ item1 item2)
         Where item1 is a pair/reference vector structure.
         Simple pattern matcher

EDIT  macro
         (EDIT item)
         Call Lispedit to edit the value of item.

EFFACE  function
         (EFFACE item list)
         Removes item from a list, updates, uses EQUAL. See page 89.

EFFACEQ  function
         (EFFACEQ item list)
         Removes item from a list, updates, uses EQ. See page 90.

ELT  function, with macro definition for compilation
         (ELT (v v0 | List | list) s-int)
         Gets an element of a list or vector, by index. See page 93, page 98 and page 94.

EMBED  function
         (EMBED id exp)
         Redefines function, may use old definition See page 143.

EMBEDDED  function
         (EMBEDDED)
         Returns list of EMBEDded functions See page 143.

ENABLE  function, with macro definition for compilation
         (ENABLE)
         Allows recognition of interrupts, cf DISABLE

ENBLSPIE  command to system interface
         (CALLBELOW 'ENBLSPIE' s-int)
         Sets up a SPIE to signal exceptions

ENVIRONVAL  function
         (ENVIRONVAL sdl sd2)
         Combine control and environment from two SDs

EGFP  function, with macro definition for compilation
         (EGFP item)
         Test a stream for end-of-file condition.
EOLP  function, with macro definition for compilation
       (EOLP item)
       Test a stream for end-of-line condition.

EQ    built in function
       (EQ item1 item2)
       Tests for identity. See page 75.

EQSUBSTLIST function
       (EQSUBSTLIST list1 list2 item)
       Replaces old items by new. Uses EQ, no update

EQSUBSTVEC function
       (EQSUBSTVEC r-vec1 r-vec2 item)
       Replaces old items by new. Uses EQ, no update

EQUAL  function
       (EQUAL item1 item2)
       Tests access equivalence. See page 76.

EQUALN function
       (EQUALN item1 item2)
       Tests access equivalence, with FUZZ = 0.0. See page 76.

ERASE function
       (ERASE file)
       Erases a disk file

ERASE command to system interface
       (CALLBELOW 'ERASE' c-str)
       Erase a file

ERRCATCH macro
       (ERRCATCH exp [id [item ...]])
       Like ERRSET, but true value, optional flag. See page 58.

ERROR  function, with macro definition for compilation
       (ERROR item)
       Inline entry to break, UNWIND forced.

ERROR-PRINT function
       (ERROR-PRINT item strm)
       Print function for break loop, = PRETTYPRINT

ERRORINSTREAM variable
       Default input stream in break loop See page 117.

ERRORN function
       (ERRORN item ...)
       Like ERROR, with multi-part message

ERROROUTSTREAM variable
       Default output stream in break loop See page 117.

ERRORR function, with macro definition for compilation
       (ERRORR item)
       Inline entry to break, FIN allowed

ERROR2 function, with macro definition for compilation
       (ERROR2 item1 item2)
       Like ERROR, two item message

ERROR3 function, with macro definition for compilation
       (ERROR3 item1 item2 item3)
       Like ERROR, three item message

ERRSET macro
       (ERRSET exp [item ...])
       Intercepts UNWINDs from inside evaluations. See page 57.

ERR1  macro
       (ERR1 s-int)
       Enters the break loop, with FIN allowed.

ERR2  macro

Table of system functions, variables and commands
(ERR2 s-int item)
Expands to a LAMBDA which enters break loop, with FIN allowed.

ERR3
macro
(ERR3 s-int)
Enter the break loop, UNWIND forced.

ERR5
function
(ERR5 s-int item)
Outline entry to break, no state saving

ERR6
function
(ERR6 s-int item app-ob)
Outline entry to break, value requested

ERR7
function
(ERR7 s-int item app-ob)
Outline entry to break, argument requested

EVAL
built in function
(EVAL exp sd)
Evaluate expression in given environment. See page 59.

EVAL-GLOBAL-ID
function
(EVAL-GLOBAL-ID id)
Global environment ID evaluator. See page 60.

EVAL-ID
function
(EVAL-ID id)
Returns fluid value of id in current stack. See page 60.

EVAL-LEX-ID
function
(EVAL-LEX-ID id)
Returns value of id, sees caller's lexicals. See page 60.

EVALANDFILEACTQ
macro
(EVALANDFILEACTQ id] exp)
Like FILEACTQ with EXF time evaluation

EVAL
built in function
(EVAL exp)
Evaluate expression in current environment. See page 59.

EVAIFUN
function
(EVAIFUN exp)
Evaluate expression, dropping lexicals See page 59.

EXF
macro
(EXF item ...)
Feed a disk file to SUPV

EXIT
built in function
(EXIT exp)
Leave a SEQ with value. See page 49.

EXP
function
(EXP num)
Exponentiation function, floats its argument. See page 111.

EXPT
function
(EXPT num1 num2)
Exponentiation function, FIX*KFIX gives FIX. See page 111.

EXTERNAL-EVENTS-CHANNELS variable
Actions for various interrupts, list

EXTSTATE
function
(EXTSTATE list1 list2 sd)
Where list1 is (id ...) and list2 (item ...). Augments the E of an SD, without capturing C.
variable
Just in case anyone expects F to be false, NIL

FASTSTREAM
function
(FASTSTREAM item)
Pragmatic test for fast io stream. See page 75.

FETCH
function
(FETCH rstrm)
Read and reconstitute an item from a LISPLIB

FETCHCHAR
function, with macro definition for compilation
(FETCHCHAR c-str s-int)
Extract a single character from a string. See page 98.

FETCHPROP
function
(FETCHPROP hashtable item1 item2)
Returns the item1 property of item2 form the hashtable.

FETCHPSMINT
function
(FETCHPSMINT c-str s-intl s-int2)
Like SUBSTRING but return a positive small integer.

FILE
definition option
(FILE rstrm)
Specifies the output LISPLIB for definition operators. See page 134.

FILEACTQ
macro
(FILEACTQ id item)
Adds expression to LISPLIB, evaluated at load

FILEIN
command to system interface
(CALLBELOW 'FILEIN' sysdep-area1 sysdep-area2 s-int)
Read a record from a file

FILELISP
function
(FILELISP file)
Save the users data on disk

FILEOUT
command to system interface
(CALLBELOW 'FILEOUT' sysdep-area1 sysdep-area2 s-int)
Write a record to a file

FILEQ
macro
(FILEQ id item)
Add an arbitrary item to a LISPLIB

FILESEG
function
(FILESEG file)
Save the shared part of the system on disk

FIN
function
(FIN [item])
A sop to APPLX, doesn’t do anything usefull

FIX
function
(FIX num)
Integer part of a floating point number. See page 105.

FIXP
built in function
(FIXP item)
Test for integerness. See page 74.

FLOAT
function
(FLOAT num)
Converts integers to floating point numbers. See page 105.

FLOATP
built in function
(FLOATP item)
Test for floatingness. See page 74.

FORCE-GLOBAL
function
(FORCE-GLOBAL id)

Table of system functions, variables and commands 157
Forces binding in proximal Global A-list

FRACODE
special form
(FRACODE exp list [lap-statement ...])
Where ARGUMENT1 is a declaration and the ARGUMENT2's are
LAP instructions.
Allows LAP code to be fed values

FRARGCOUNT
function
(FRARGCOUNT built-in-function)
Minimum number of arguments for FRs

FREEZE-SHARED-SEGMENT
function
(FREEZE-SHARED-SEGMENT)
Makes the shared segment area out of bounds

FRP
built in function
(FRP item)
Tests for FR type. See page 73.

FUNARG
special form
(FUNARG exp sd)
Creates a funarg object from it's arguments. See page 55.

FUNARGP
built in function
(FUNARGP item)
Test for FUNARG type. See page 75.

FUNCTION
special form
(FUNCTION exp)
Makes a funarg out of an unevaluated argument. See page 55.

GCCOUNT
function
(GCCOUNT)
Number of times RECLAIM has been called

GCMMSG
function
(GCMMSG boolean)
Turns on and off RECLAIMs statistic message

GCNLIST
command to system interface
(CALLBELOW 'GCNLIST' C-s-tri ('INPUT' | 'OUTPUT')
sysdep-area2)
Create a console I/O PLIST

GENLABEL
function
(GENLABEL)
Creates a GENSYM based on ,LABELNUM. See page 114.

GENSYM
function, with macro definition for compilation
(GENSYM)
Creates a GENSYM based on the system seed. See page
undefined refid=gensym..

GENSYMP
built in function
(GENSYMP item)
Tests for gensym type. See page 73.

GET
function
(GET (id | list) item)
Searches a property or name-value list. Set page 115 and
page 86.

GETBITSTR
function
(GETBITSTR s-int)
Creates a new bitstring. See page 95.

GETCH
function
(GETCH C-s-tri s-int)
Synonym for FETCHCHAR

GETFLT
function
(GETFLT)

158 YKTLISP Program Description and Operations Manual
Create a new floating point number cell. See page undefined.

**GETFULLSTR**

*function*  
(GETFULLSTR s-int [[]id | c-str | s-int]])  
Create a string, filled with a given character. See page 95.

**GETHASHARRAY**

*function*  
(GETHASHARRAY s-int)  
Create a hasharray with an initial size of (EXPT 2 s-int) entries.

**GETHASHSTRING**

*function*  
(GETHASHSTRING c-str hash-array)  
String to hash table entry/index

**GETREALV**

*function*  
(GETREALV s-int)  
Create a vector of floating point numbers. See page 91

**GETREFV**

*function*  
(GETREFV s-int)  
Create a vector of s-exps. See page 91

**GETSTR**

*function*  
(GETSTR s-int)  
Create an empty string with given capacity. See page 95.

**GETWORDV**

*function*  
(GETWORDV s-int)  
Create a vector of integers (32 bits). See page 91

**GETZEROVEC**

*function*  
(GETZEROVEC s-int)  
Create integer vector filled with zeros. See page 91

**GFIPLIST**

*command to system interface*  
(CALLBELOW 'GFIPLIST' c-str1 ('INPUT' | 'OUTPUT') c-str2 s-int sysdip-area3)  
Create a file I/O PLIST

**GGREATERP**

*function*  
(GGREATERP item1 item2)  
Generic comparison, arbitrary order of types

**GLOEXTSTATE**

*function*  
(GLOEXTSTATE list1 list2 sd)  
Augments GLUE of an SD, without capturing C.

**GO**

*special form*  
(GO label)  
Transfer control to a label in a SEQ. See page 49.

**GREATERP**

*function*  
(GREATERP num1 num2)  
Compare two numbers. See page 116.

**HASHARRAYP**

*function*  
(HASHARRAYP item)  
Value is argument if it is a hasharray.

**HASHINIT**

*function*  
(HASHINIT)  
Discard all existing hasharrays.

**HASHITEM**

*function*  
(HASHITEM hasharray item1 item2)  
Make an entry into a hasharray for item1 with value item2.

**HASHPROP**

*function*  
(HASHPROP hasharray item1 item2 item3)  
Set item2 property of item1 to item3 in the hasharray.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAPLEFT</td>
<td>function (HEAPLEFT) Returns bytes of HEAP space remaining</td>
</tr>
<tr>
<td>HEXEXP</td>
<td>function (HEXEXP item) Convert a LISP pointer to a hexadecimal string</td>
</tr>
<tr>
<td>HEXNUM</td>
<td>function (HEXNUM fx-num) Convert an integer to a hexadecimal string</td>
</tr>
<tr>
<td>HEXSTRINGPART</td>
<td>function (HEXSTRINGPART c-str s-int1 s-int2) Substring to hexadecimal, 1-2-3 or 4 chars</td>
</tr>
<tr>
<td>HPROPLIST</td>
<td>function (HPROPLIST hasarray item) Fetch an entire entry from the hasarray (does not include the key)</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>function, with macro definition for compilation (IDENTITY item) (LAMBDA (X) X). See page 69.</td>
</tr>
<tr>
<td>IDENTP</td>
<td>built-in function (IDENTP item) Tests for identifier type. See page 73.</td>
</tr>
<tr>
<td>IFCAR</td>
<td>function, with macro definition for compilation (IFCAR item) If argument is a pair, CAR, otherwise NIL. See page 78.</td>
</tr>
<tr>
<td>IFCDR</td>
<td>function, with macro definition for compilation (IFCDR item) If argument is a pair, CDR, otherwise NIL. See page 78.</td>
</tr>
<tr>
<td>INITIALOPEN</td>
<td>function (INITIALOPEN) Does a fixup of the globally bound streams</td>
</tr>
<tr>
<td>INITSUPV</td>
<td>variable NIL or expression to start user's supervisor</td>
</tr>
<tr>
<td>INITSYMTAB</td>
<td>definition option (INITSYMTAB . &amp;a-list.) Provided a pre-defined set of EQUs for ASSEMBLE. See page 133.</td>
</tr>
<tr>
<td>INTERN</td>
<td>function (INTERN c-str) String to identifier, copies pname. See page 113.</td>
</tr>
<tr>
<td>INTERSECTION</td>
<td>function (INTERSECTION list1 list2) Set intersection of two lists, no duplicates. See page 83.</td>
</tr>
<tr>
<td>INTERSECTIONQ</td>
<td>function (INTERSECTIONQ list1 list2) Set intersection, uses EQ, no duplicates. See page 83.</td>
</tr>
<tr>
<td>IOSTATE</td>
<td>function (IOSTATE file-name) Tests for the existence of a disk file</td>
</tr>
<tr>
<td>IOSTATEQ</td>
<td>function (IOSTATEQ file-name) Tests for the writeability of a disk file</td>
</tr>
<tr>
<td>IS-CONSOLE</td>
<td>function (IS-CONSOLE item) Tests a stream for (DEVICE . CONSOLE) property. See page 75.</td>
</tr>
</tbody>
</table>
ITEM-N-ADV function, with macro definition for compilation  
(ITEM-N-ADV strm)  
Get current stream item and step to the next. See page  
undefined.

L-CASE function  
(L-CASE (c-str | id | list))  
Shifts string, id, or list thereof to 1 case. See page  
102 and page 114.

LAM macro  
(LAM bv-list [exp ...])  
Extension of the LAMBDA special form

LAMBDA special form  
(LAMBDA bv-list [exp ...])  
Special form for applicable expressions. See page 52.

LAPLIST definition option  
(LAPLIST . boolean)  
Requests a listing of the assembler code. See page 133.

LAST function  
(LAST list)  
Final item of a list (ignoring final CDR). See page 84.

LAST-EXP function  
(LAST-EXP)  
Returns the last expression SUPV evaluated

LAST-VALUE function  
(LAST-VALUE)  
Returns the last value computed by SUPV

LASTNODE function  
(LASTNODE list)  
Final pair of a list. See page 84.

LEFTSHIFT function  
(LEFTSHIFT num s-int)  
Multiply by a power of two. See page 110.

LENGTH function  
(LENGTH item)  
Number of items in a list. See page 90.

LENGTHCODE function  
(LENGTHCODE vec)  
Number of bytes occupied by a vector. See page 93 and  
page 98.

LESSP function  
(LESSP num1 num2)  
GREATERP, backwards. See page 106.

LETTERIZER variable  
Global whose value defines the letterizer, |  

LINTP built-in function  
(LINTP item)  
Tests for long integer type. See page 74.

LISPRET command to system interface  
(CALLBELOW 'LISPRET' s-int sysdep-area)  
Return control to invoking environment

LISPSEG command to system interface  
(CALLBELOW 'LISPSEG')  
Returns address of LISP shared segment.

LIST function, with macro definition for compilation  
(LIST [item ...])  
Makes a list of its arguments. See page 81.

LIST-PAIRS macro  

Table of system functions, variables and commands  

(LIST-PAIRS &EXP)
Like COUNT-PAIRS but prints a file CALL PAIRS A.

LISTING
definition option
(LISTING . strm)
Provides a stream for program listings. See page 134.

LISTOFFLUIDS
function
(LISTOFFLUIDS bpi)
Returns variables bound FLUID by a BPI

LISTOFFREES
function
(LISTOFFREES bpi)
Returns variables used free by a BPI

LISTOFFFUNCTIONS
function
(LISTOFFFUNCTIONS bpi)
Returns names of functions called by a BPI

LISTOFFLEXICALS
function
(LISTOFFLEXICALS bpi)
Returns variables bound LEXically by a BPI

LISTOFFQUOTES
function
(LISTOFFQUOTES bpi)
Returns all items QUOTEd by a BPI

LISTOFFSAME
function
(LISTOFFSAME list)
Tests for identical types on list members

LISTP
built in function
(LISTP item)
Test for listness, i.e. pair or NIL. See page 72.

LIST2FLTVEC
function
(LIST2FLTVEC list)
Where all elements of list are numbers.
Converts a list to a vector of floating nums. See page 91.

LIST2IVEC
function
(LIST2IVEC list)
Where all elements of list are fixed point numbers.
Converts a list to a vector of integers. See page 92.

LIST2REFVEC
function
(LIST2REFVEC list)
Converts a list to a vector of s-exps. See page 91.

LN
function
(LN num)
Natural logarithm. See page 111.

LOADCOND
function
(LOADCOND file)
Loads currently undefined items from a LISPLIB. See page 127.

LOADVOL
function
(LOADVOL file)
Loads all items from a LISPLIB. See page 127.

LOG
function
(LOG NUM)
Logarithm, base 10. See page 111.

LOG2
function
(LOG2 NUM)
Logarithm, base 2. See page 111.

LOTSOF
function
(LOTSOF item ...)
Returns "infinite list" of arguments
MACRO-APP-SD  
function       (MACRO-APP-SD file)  
Adds items from a LIBRARY to MACRO-APP-SD. See page 132.

MAKEPROP       
function       (MAKEPROP id | list | item)  
Add or change a name-value property. See page 115 and  
pages 75-78.

MAKESTRING     
function, with macro definition for compilation  
(MAKESTRING c-str)  
Returns c-str arg, compiles with data inline. See page  
97.

MAKETRSTRING   
function       (MAKETRSTRING c-str | list)  
Builds translate and test tables. See page 97.

MAP            
macro          (MAP list app-ob)  
Equivalent to MMAP with arguments reversed. See page 68.

MAPCAR         
macro          (MAPCAR list app-ob)  
Equivalent to MMAPCAR with arguments reversed. See page 68.

MAPE           
macro          (MAPE app-ob (vec | list) ...)  
Applies operator to vector elements, returns size. See  
page 67.

MAPELT         
macro          (MAPELT app-ob (vec | list) ...)  
Applies operator to vector elements, returns vec. See  
page 67.

MAPLIST        
macro          (MAPLIST list app-ob)  
Equivalent to MMAPLIST with arguments reversed. See page  
68.

MAPOBLIST      
function       (MAPOBLIST app-ob)  
Applies function to elements of obarray. See page 68.

MAPSET       
macro          (MAPSET app-ob (vec | list) ...)  
Applies operator to vector elements, SETELTs list v. See  
page 67.

MASKNUM        
function       (MASKNUM fx-num s-int)  
Rightmost n bits of a number

MATEMPDEVICE   
function       (MATEMPDEVICE list1 [list2])  
Where item is either a name/expression list or a list of  
names/expression lists.  
Adds definitions to MACRO-APP-SD. See page 132.

MATEMPSETQ     
function       (MATEMPSETQ id item [list])  
Adds values to MACRO-APP-SD

MAX            
function, with macro definition for compilation  
(MAX num ... )  
Largest of a set of numbers. See page 106.

MAXINDEX       
function

Table of system functions, variables and commands
(MAXINDEX vec)
Largest valid index, = (SUB1 (SIZE x)). See page 92.

MDEF
built in function
(MDEF macro-app-ob item ed)
MDEFX with respect to an environment

MDEFX
built in function
(MDEFX macro-app-ob item)
Single level macro expansion

MDO
macro
(MDO list1 list2 [exp ...])
Where list1 declare variable, with initial and subsequent values, and list2 defines the termination test, epilogue and value.
MacLisp based iterator.

MEMBER
function
(MEMBER item list)
Membership of item in list, uses EQUAL. See page 85.

MEMQ
function
(MEMQ item list)
Membership of item in list, uses EQ. See page 85.

MESSAGE
definition option
(MESSAGE . strm)
Provides a stream for information and warning messages. See page 134.

MIN
function, with macro definition for compilation
(MIN num ...)
Smallest of a set of numbers. See page 107.

MINUS
function
(MINUS num)
Changes the sign of a number. See page 107.

MINUSP
function
(MINUSP item)
Tests for number less than zero. See page 106.

MLAMBDA
special form
(MLAMBDA by-list [exp ...])
Special form for macro applicable expressions. See page 53.

MMAP
macro
(MMAP app-ob list ...)
Apply function to succeeding CDRs. See page 64.

MMAPC
macro
(MMAPC app-ob list ...)
Apply function to succeeding items. See page 64.

MMAPCAH
macro
(MMAPCAH app-ob list ...)
Apply function to succeeding items, HCONC values. See page 65.

MMAPCAR
macro
(MMAPCAR app-ob list ...)
Apply function to succeeding items, LIST values. See page 64.

MMAPCON
macro
(MMAPCON app-ob list ...)
Apply function to succeeding CDRs, NCONC values. See page 65.

MMAPLACA
macro
(MMAPLACA app-ob list ...)
Apply function to succeeding CARs, RPLACA 1st arg.. See page 65.
macro
(MMAPLIST app-ob list ...)
Apply function to succeeding CDRs, LIST values. See page 64.

MONITOR
function
(MONITOR id [list1 [list2]])
Sets a function to print on call and return. See page 141.

MONITOR-ON-OFF
variable
Global switch to allow monitoring PRINT et al.

MOVEVEC
function
(MOVEVEC vec1 vec2)
Copies contents of one vector into another. See page 94.

MRP
built in function
(MRP item)
Tests for MR. See page 73.

MSUBRP
built in function
(MSUBRP item)
Tests for MSUBR type. See page 72.

MTON
function
(MTON s-int1 s-int2)
Creates list of integers from n to m. See page 83.

NAMEDERRSET
macro
(NAMEDERRSET id exp [item ...])
ERRSET with tag usable by THROW. See page 58.

NCONC
function
(NCONC list1 list2)
APPENDs by updating, no copying. See page 88.

NCONSTKD
command to system interface
(CALLBELOW 'NCONSTKD')
Returns no. of physical lines in console stack

ENABLE
command to system interface
(CALLBELOW 'ENABLE' s-int1 s-int2 s-int3 s-int4)
= ENABLE; for versions after 0023

NEWAREA
command to system interface
(CALLBELOW 'NEWAREA' s-int)
Allocates FREE/FRED storage.

NEWQUEUE
function
(NEWQUEUE s-int1 s-int2)
Changes max queue length for an interrupt

NEXT
function, with macro definition for compilation
(NEXT strm)
Advance an input stream one item. See page 119.

NILFN
function, with macro definition for compilation
(NILFN)
(LAMBDA () ()). See page 69.

NILLEFT
function
(NILLEFT)
Returns bytes of NIL space remaining

NILSD
function
(NILSD item)
Returns the SD for the root of the stack

NOLINK
definition option
(NOLINK . boolean)
Controls the creation of a bp following assembly. See page 132.

NONINTEERRUPTIBLE
definition option

Table of system functions, variables and commands
NONINTERRUPTIBLE boolean
Controls the compilation of interrupt polling instructions. See page 133.

NONSTOREDP
function, with macro definition for compilation
(NONSTOREDP item)
Tests for nonstored (i.e., not heap resident). See page 71.

NOT
function, with macro definition for compilation
(NOT item)
True in argument is NIL, NIL otherwise. See page 71.

NOTEFILE
function
(NOTEFILE strm)
Returns position information for file stream

NREVERSE
function
(NREVERSE list)
Reverses a list in place by updating. See page 89.

NSTACKED
command to system interface
(CALLBELLOU 'NSTACKED')
Returns number of lines in program stack

NSUBST
function
(NSUBST item1 item2 item3)
Update a structure with item substitutions

NTUPLEP
built-in function
(NTUPLEP item)
Tests for ntuple type

NULL
built-in function
(NULL item)
Tests for NIL value. See page 71.

NULLOUTSTREAM variable
A no-op stream, a data sink See page 117.

NUMBEROFARGS
function
(NUMBEROFARGS bpi)
Returns number of argument for a BPI

NUMBERP
built-in function
(NUMBERP item)
Tests for number types. See page 74.

NUM2TIME
function
(NUM2TIME fx-num)
Unpacks small int. to time/date (see TIME2NUM)

OBARRAY
function
(OBARRAY)
Returns copy of the obarray. See page 116.

OBDUMP
function
(OBDUMP id)
Evaluate id and prettyprint (id value) if the value is non-trivial.

OBEY
function
(OBEY c-str)
Passes a command to the host system

OBEY
command to system interface
(CALLBELLOU 'OBEY' c-str)
Pass a command string to the system

ODDP
function
(ODDP item)
Odd test for integers. See page 106.

ONE-OF
macro
((ONE-OF id1 ... idn) idn)
Test for arg EQ to member of QUOTEd set

**OP-RECOGNITION-SD** definition option
(OP-RECOGNITION-SD . sd)
Specifies operator recognition environment for
definition. See page 132.

**OPTIMIZE**
definition option
(OPTIMIZE . s-int)
Controls the amount of optimization by the compiler. See
page 134.

**OPTIONLIST**
variable
A-list which controls DEFINE's actions

**OR**
macro
(OR [exp ...])
Evaluates expressions until one returns non-NIL. See
page 51

**ORADDTEMPDEFS**
function
(ORADDTEMPDEFS file)
Adds items from a LIBRARY to OP-RECOGNITION-SD See page
131.

**ORBIT**
function
(ORBIT b-str ...)
Logical or of bitstrings. See page 97.

**ORTEMMPDEF**
function
(ORTEMMPDEF list1 [list2])
Where item is either a name/expression list or a list of
name/expression lists.
Adds definitions to OP-RECOGNITION-SD See page 132.

**ORTEMPSAT**
function
(ORTEMPSAT id item [list])
Adds values to OP-RECOGNITION-SD

**PACKHEXSTRING**
function
(PACKHEXSTRING c-str)
Creates string whose hex form is argument

**PAIRP**
built in function
(PAIRP item)
Tests for pair type. See page 72.

**PANICMSG**
command to system interface
(CALLBELOW 'PANICMSG' c-str)
Writes to console unconditionally

**PARAMETERS**
function
(PARAMETERS)
Returns the parameter string from LISP370

**PARMLIST**
command to system interface
(CALLBELOW 'PARMLIST' sysdep-area)
Retrieves the parameter list from invocation

**PLACEP**
function, with macro definition for compilation
(PLACEP item)
Tests for read place holder type. See page 74.

**PLEXP**
built in function
(PLEXP item)
Tests for plex type

**PLUS**
function, with macro definition for compilation
(PLUS num ...)
Sums a set of numbers. See page 107.

**PLUSEP**
function
(PLUSEP item)
Tests for positive numbers. See page 105.

Table of system functions, variables and commands 167
function
(PNAME id)
Returns a copy of the p-name of an id. See page 114 and page 97.

POINTFILE
function
(POINTFILE pair strm)
Repositions a file stream

POLLUP
function, with macro definition for compilation
(POLLUP)
Test for upward branch interrupt

macro
(POP id1 [id2])
Pops an item from a list. Value is the new CDR, which becomes value to id1. Optionally assigns CAR to id2.

POPP
macro
(POPP id1 [id2])
Pop an item from a list, setting id1 to new list. Value is the old list. Optionally assigns CAR to id2.

POST
function
(POST s-int item)
Signal an interrupt

POST-SELECT
function
(POST-SELECT s-int1 item s-int2)
Signals an interrupt, enables selected POLL

PRETTYPRINT
function, with macro definition for compilation
(PRETTYPRINT item [strm])
Formatted print, with TERPRI. See page 123.

PRETTYPRINTF
function, with macro definition for compilation
(PRETTYPRINTF item [strm])
Formatted print, no TERPRI. See page 123.

PRINM
function, with macro definition for compilation
(PRINM list [strm])
Print list elements, may default stream. See page 123.

PRINT
function, with macro definition for compilation
(PRINT item [strm])
Print item and TERPRI, may default stream. See page 123.

PRINTCH
function, with macro definition for compilation
(PRINTCH char [strm])
Put character on a stream, may default stream. See page 121.

PRINTEXP
function, with macro definition for compilation
(PRINTEXP c-str [strm])
Print contents of string, may default stream. See page 121.

PRINTVAL
function
(PRINTVAL c-str strm)
Print routine for SUPV

PRINTWARM
function
(PRINTWARM list)
Does PRINM on CURDOUTSTREAM, forcing new line

PRINO
function, with macro definition for compilation
(PRINO item [strm])
Print item, may default stream. See page 122.

PRINOR
function
(PRINOR item s-int [strm])
Print item right justified in a field of width smint.

PRIN1
function, with macro definition for compilation
(PRIN1 item [strm])

168 YKLILSP Program Description and Operations Manual
Print atom, may default stream. See page 122.

PRINLB

function, with macro definition for compilation
(PRINLB item [strm])
Print atom + blank, may default stream. See page 122.

PROG

macro
(PROG list [[exp | id] ...])
Bind, initialize variables, establish labels. See page undefined.

PROGN

special form
(PROGN [exp ...])
Evaluate expressions in seq., value is last. See page 48.

PROGRAM-EVENTS

variable
A-list of actions on errors, list

PROG1

function, with macro definition for compilation
(PROG1 exp ...) Evaluate expressions in seq., value is first. See page 48.

PROG2

function, with macro definition for compilation
(PROG2 exp exp ...) Evaluate expressions in seq., value is second. See page 48.

PROPLIST

function
(PROPLIST id)
Returns id's property list, partial copying. See page 115.

PRY

function
(PRY [item ...]) Forces entry to machine debugger

PSMINTP

macro
(PSMINTP item)
Equivalent to QSPLUSP.

PUSH

macro
(PUSH item id)
CONS the value of item onto a list, value of id. Sets id to new list.

PUTBACK

function
(PUTBACK item strm)
Replaces item onto head of input stream. See page 121.

QASSQ

function, with macro definition for compilation
(QASSQ item list) ASSQ, functional version of in-line macro. See page 86.

QCAAAAR

function, with macro definition for compilation
(QCAAAAR item)
(QCAR (QCAR (QCAR (QCAR x)))) See page 78.

QCAAADR

function, with macro definition for compilation
(QCAAADR item)
(QCAR (QCAR (QCAR (QCAR x)))) See page 78.

QCAAR

function, with macro definition for compilation
(QCAAR item)
(QCAR (QCAR (QCAR x))) See page 78.

QCAADR

function, with macro definition for compilation
(QCAADR item)
(QCAR (QCAR (QCAR (QCAR x)))) See page 78.

QCAADDR

function, with macro definition for compilation
(QCAADDR item)
(QCAR (QCAR (QCAR (QCDR x)))) See page 78.

Table of system functions, variables and commands 169
QCAADR  function, with macro definition for compilation
         (QCAADR item)
         (QCAR (QCAR (QCDR x))). See page 78.

QCAAR   function, with macro definition for compilation
         (QCAAR item)
         (QCAR (QCAR x)). See page 78.

QCADAAR function, with macro definition for compilation
         (QCADAAR item)
         (QCAR (QCDR (QCAR (QCAR x))). See page 78.

QCADADR function, with macro definition for compilation
         (QCADADR item)
         (QCAR (QCDR (QCAR (QCDR x))). See page 78.

QCADAR  function, with macro definition for compilation
         (QCADAR item)
         (QCAR (QCDR (QCAR x))). See page 78.

QCADDAR function, with macro definition for compilation
         (QCADDAR item)
         (QCAR (QCDR (QCAR (QCDR x))). See page 78.

QCADDAR function, with macro definition for compilation
         (QCADDAR item)
         (QCAR (QCDR (QCDR (QCDR x))). See page 78 and page 84.

QCADDR function, with macro definition for compilation
         (QCADDR item)
         (QCAR (QCDR (QCDR x))). See page 78 and page 84.

QCADR  function, with macro definition for compilation
         (QCADR item)
         (QCAR (QCDR x)). See page 78 and page 84.

QCAR    function, with macro definition for compilation
         (QCAR item)
         Like CAR, but no argument type check made. See page 78
         and page 84.

QCDAALAR function, with macro definition for compilation
         (QCDAALAR item)
         (QCDR (QCAR (QCAR (QCAR x))). See page 78.

QCDAAR  function, with macro definition for compilation
         (QCDAAR item)
         (QCDR (QCAR (QCAR x))). See page 78.

QCDAARD function, with macro definition for compilation
         (QCDAARD item)
         (QCDR (QCAR (QCAR x))). See page 78.

QCDAADR function, with macro definition for compilation
         (QCDAADR item)
         (QCDR (QCAR (QCAR x))). See page 78.

QCDDAR function, with macro definition for compilation
         (QCDDAR item)
         (QCDR (QCAR (QCAR x))). See page 78.

QCDDAAR function, with macro definition for compilation
         (QCDDAAR item)
         (QCDR (QCDR (QCAR (QCAR x))). See page 78.

QCDDADR function, with macro definition for compilation
         (QCDDADR item)
(QCDR (QCDR (QCAR (QCDR x)))). See page 78.

QCDRAR  
function, with macro definition for compilation  
(QCDRAR (QCDRAR item))  
(QCDR (QCDR (QCAR x))). See page 78.

QCDDDAR  
function, with macro definition for compilation  
(QCDDDAR item)  
(QCDR (QCDR (QCAR x))). See page 78.

QCDDDDAR  
function, with macro definition for compilation  
(QCDDDDAR item)  
(QCDR (QCDR (QCAR x))). See page 78 and page 84.

QCDDDDAR  
function, with macro definition for compilation  
(QCDDDDAR item)  
(QCDR (QCDR (QCAR x))). See page 78 and page 84.

QCDR  
function, with macro definition for compilation  
(QCDR item)  
Like CDR, but no argument type check made. See page 78 and page 84.

QDCQ  
macro  
(QDCQ item1 item2)  
Where item1 is a pair/reference vector structure.  
Like DCQ, but no type checking.

QEAPPEND  
macro  
((QEAPPEND (id | c-str | s-int) ...) c-str2)  
SUFFIXes QUOTEd chars to c-str2, no check.

QECHAR  
macro  
((QECHAR s-int) c-str)  
Extracts char from c-str, no check.

QECHARN  
function, with macro definition for compilation  
((QECHARN s-int) c-str)  
Extracts char code from c-str, no check.

QECQ  
macro  
(QECQ item1 item2)  
Where item1 is a pair/reference vector structure.  
Like ECQ, but no type checking.

QEFILL  
macro  
((QEFILL (id | c-str | s-int)) c-str2)  
Pads c-str2 with QUOTEd character, no check.

QESTORE  
macro  
((QESTORE s-int (id | c-str | s-int) ...) c-str2)  
Replaces chars in c-str2 by QUOTEd chars.

QESUFFN  
macro  
(QESUFFN c-str s-int)  
Suffix a character to a string.

QETEST1  
macro  
((QETEST1 s-int (id | c-str | s-int) ...) c-str2)  
Test for char in c-str2, no check.

QGET  
function, with macro definition for compilation  
(QGET (id | list) item)  
GET, functional version of in-line macro. See page undefined and page 86.

QHIGHHALF  
function, with macro definition for compilation  
(QHIGHHALF flt)  
First 4 bytes of floating point number as fixed

QINSERT  
function, with macro definition for compilation  

Table of system functions, variables and commands 171
(QINSERT s-int1 s-int2 c-str fx-num)
Insert bytes from fixed number into string

QINSERTFP
function, with macro definition for compilation
(QINSERTFP s-int c-str flt)
Insert floating point number into string

QINSERTSTG
function, with macro definition for compilation
(QINSERTSTG s-int c-str1 c-str2)
Insert string into string

QLLENGTH
function, with macro definition for compilation
(QLLENGTH list)
Inline LENGTH, no check for cycle. See page undefined.

QLLENGTHCODE
function, with macro definition for compilation
(QLLENGTHCODE vec)
Inline LENGTHCODE, no type check. See page 93 and page 98.

QLLOWHALF
function, with macro definition for compilation
(QLLOWHALF flt)
Last 4 bytes of floating point number as fixed

QMEMQ
function, with macro definition for compilation
(QMEMQ item list)
MEMQ, functional version of in-line macro. See page 85.

QRCQ
macro
(QRCQ item1 item2)
Where item2 is a pair/reference vector structure.
Like RCQ, but no type checking.

QREFELT
function, with macro definition for compilation
(QREFELT r-vec s-int)
Like ELT of reference vector, no type check. See page 94.

QREFVECLENGTH
function, with macro definition for compilation
(QREFVECLENGTH r-vec)
SIZE for reference vectors, no checking. See page 93.

QREFVECMAXINDEX
function, with macro definition for compilation
(QREFVECMAXINDEX r-vec)
MAXINDEX for reference vectors, no checking. See page 93.

QRPLACA
function, with macro definition for compilation
(QRPLACA pair item)
Like RPLACA, no type check. See page 78 and page 88.

QRPLACAD
function, with macro definition for compilation
(QRPLACAD pair1 pair2)
Like RPLACAD, no type check. See page 79.

QRPLACD
function, with macro definition for compilation
(QRPLACD pair item)
Like RPLACD, no type check. See page 79 and page 88.

QRPLNODE
function, with macro definition for compilation
(QRPLNODE pair item1 item2)
Like RPLNODE, no type check. See page 79.

QSABSVAL
function, with macro definition for compilation
(QSABSVAL s-int)
Like ABSVAL of small integer, no type check. See page 107.

QSADD1
function, with macro definition for compilation
(QSADD1 s-int)
Like ADD1 of small integer, no type check. See page 108.

QSAND
function, with macro definition for compilation
(QSAND s-int1 s-int2)
Bitwise AND of small integers, no check. See page 110.
QSBITS macro
((QSBITS s-int1 s-int2) s-int3)
Extracts bits from small integer, no check

QSCANGELENGTH function, with macro definition for compilation
(QSCANGELENGTH s-str s-int)
Like CHANGELENGTH, no type check. See page 101.

QSDECL function, with macro definition for compilation
(QSDECL s-int)
Like QSSUBL, but won’t cross zero. See page 108.

QSDIFFERENCE function, with macro definition for compilation
(QSDIFFERENCE s-int1 s-int2)
Like DIFFERENCE of small integers, no check. See page 108.

QSENBITS macro
((QSENBITS s-int1 s-int2) s-int3 s-int4)
Sets bits in small integer, no check

QSETRFV function, with macro definition for compilation
(QSETRFV f-vec s-int item)
Like SETELT of reference vector, no check. See page 94.

QSGREATERP function, with macro definition for compilation
(QSGREATERP s-int1 s-int2)
Like GREATERP of small integers, no check. See page 106.

QSMINC function, with macro definition for compilation
(QSMINC s-int)
Like QSADD1, but won’t cross zero. See page 108.

QSELFSHIFT function, with macro definition for compilation
(QSELFSHIFT s-int1 s-int2)
Like LEFTSHIFT of small integers, no check. See page 110.

QSLESSP function, with macro definition for compilation
(QSLESSP s-int1 s-int2)
Like LESSP of small integers, no check. See page 106.

QSMAX function, with macro definition for compilation
(QSMAX s-int1 s-int2)
Like MAX of small integers, no check. See page 107.

QSMIN function, with macro definition for compilation
(QSMIN s-int1 s-int2)
Like MIN of small integers, no check. See page 107.

QSMINUS function, with macro definition for compilation
(QSMINUS s-int)
Like MINUS of small integers, no check. See page 107.

QSMINUSP function, with macro definition for compilation
(QSMINUSP s-int)
Like MINUSP of small integers, no check. See page 106.

QSNOT function, with macro definition for compilation
(QSNOT s-int)
Bitwise NOT of small integer, no check. See page 110.

QSODDP function, with macro definition for compilation
(QSODDP s-int)
Like ODDP of small integer, no check. See page 106.

QSGOR function, with macro definition for compilation
(QSGOR s-int1 s-int2)
Bitwise OR of small integers, no check. See page 110.

QSORT function
(QSORT list)
Quick sort on lists, uses SORTGREATERP. See page 90.

QSPLUS function, with macro definition for compilation
(QSPLUS s-int1 s-int2)
Like PLUS of small integers, no check. See page 107.

QSPLUSP
function, with macro definition for compilation
(QSPLUSP s-int)
Like PLUSP of small integers, no check. See page 105.

QSQUOTIENT
function, with macro definition for compilation
(QSQUOTIENT s-int1 s-int2)
Like QUOTIENT of small integers, no check. See page 109.

QSREMAINDER
function, with macro definition for compilation
(QSREMAINDER s-int1 s-int2)
Like REMAINDER of small integers, no check. See page 110.

QSSUBL
function, with macro definition for compilation
(QSSUBL s-int)
Like SUB1 of small integers, no check. See page 108.

QSTIMES
function, with macro definition for compilation
(QSTIMES s-int1 s-int2)
Like TIMES of small integers, no check. See page 109.

QSTRIM
macro
((QSTRIM s-int c-str))
Inline CHANGELENGTH, no check

QSTRINGLENGTH
function, with macro definition for compilation
(QSTRINGLENGTH c-str)
Like SIZE for character strings, no check. See page undefined.

QSXOR
function, with macro definition for compilation
(QSXOR s-int1 s-int2)
Bitwise XOR of small integers, no check. See page 111.

QSZEROP
function, with macro definition for compilation
(QSZEROP s-int)
Like ZEROP of small integers, no check. See page 105.

QUIET
definition option
(QUIET . boolean)
Suppresses messages to the console. See page 134.

QUOTE
special form
(QUOTE item)
Returns argument un-evaluated. See page 47.

QUOTEIZER
variable
Global whose value defines the quoteizer, "

QUOTIENT
function, with macro definition for compilation
(QUOTIENT num1 num2)
(CAR (DIVIDE x y)), quotient after division. See page 109.

RANDOM
function
(RANDOM)
Return a random 32-bit integer based on the system seed.

RANDOMCJS
function
(RANDOMCJS &int.)
Return a random 32-bit integer based on the 32-bit input.

RCLASS
function
(RCLASS c-str pstrm)
Returns the class byte from LISPLIB item. See page 125.

RCOPYITEMS
function
(RCOPYITEMS file1 file2 list)
Copy item(s) from one LISPLIB to another. See page 127.

RCQ
macro
(RCQ item1 item2)
Where item2 is a pair/reference vector structure.
Updates components of structure from ID values

RDCHR function, with macro definition for compilation
(RDCHR [strm])
Like ITEM-AND-ADVANCE, may default stream. See page 119.

RDEFI STREAM function
(RDEFI STREAM list)
Create a stream for LISLIB operations. See page 125.

RDL I NE command to system interface
(CALLBELOW 'RDL I NE' sysdep-areal sysdep-area2)
Reads line from console

RDROPITEMS function
(RDROPITEMS file list)
Delete item(s) from a LISLIB file. See page 127.

READ function, with macro definition for compilation
(READ [strm])
Reads any s-expression, may default stream. See page 120.

READ-LINE function
(READ-LINE strm)
Reads a line (record). See page 119.

READPLACEGEN function
(READPLACEGEN)
Create a read place holder

REALVECP built in function
(REALVECP item)
Test for real vector type. See page 72.

RECLAIM function
(RECLAIM)
The garbage collector

REFVECP built in function
(REFVECP item)
Test for reference vector type. See page 72.

RELPAGES command to system interface
(CALLBELOW 'RELPAGES' s-int1 s-int2)
Release virtual pages

REMAINDER function, with macro definition for compilation
(REMAINDER num1 num2)
(CADR (DIVIDE x y)), remainder after division. See page 103.

REMALLPROPS function
(REMALLPROPS id)
Remove all items from a property list. See page 116.

REMOVE function
(REMOVE list item [s-int])
Delete one (or s-int) occurrence(s) of item in list.
Uses EQUAL and copies input list.

REMOVEQ function
(REMOVEQ list item [s-int])
Delete one (or s-int) occurrence(s) of item in list.
Uses EQ and copies input list.

REMOV E R function
(REMOVER list item [s-int])
Delete one (or s-int) occurrence(s) of item in list.
Uses EQUAL and updates input list.

REMOVEQR function
(REMOVEQR list item [s-int])

Table of system functions, variables and commands 175
Delete one (or s-int) occurrence(s) of item in list.  
Uses EQ and updates input list.

REMPROP
function
(REMPROP id item)
Remove a specific item from a property list. See page
115 and page undefined.

RENAME
command to system interface
(CALLEBLOW 'RENAME' sysdep-area1 sysdep-area2)
Rename a disk file

REPLACE
command to system interface
(CALLEBLOW 'REPLACE' sysdep-area1 sysdep-area2)
Safe rename for disk files

REPLACEFILE
function
(REPLACEFILE file-name1 file-name2)
Functional link to CALLBELOW REPLACEF

RESETQ
macro
(RESETQ id [item])
Assigns to a variable, returns previous value. See page
61.

RESOLVEF
command to system interface
(CALLEBLOW 'RESOLVEF' sysdep-area1 sysdep-area2)
Returns data for resolved file

RESTARTSD
variable
Resume point on warm start, SD

RET
function
(RET [s-int [sysdep-area]])
Exit to host system

RETURN
built in function
(RETURN item)
Exit from a LAMBDA contour with value. See page 49.

REVERSE
function
(REVERSE list)
Reverses order of a list, no updating. See page 82.

RIGHTSHIFT
function, with macro definition for compilation
(RIGHTSHIFT num s-int)
Divide by a power of two. See page undefined.

RKEYIDS
function
(RKEYIDS file)
Identifiers matching LISPLIB keys. See page 126.

RPACKFILE
function
(RPACKFILE file)
Garbage collect a LISPLIB file. See page 126.

RPLACA
built in function
(RPLACA pair item)
Update the first element of a pair. See page 78 and page
87.

RPLACAD
function
(RPLACAD pair1 pair2)
Replace CAR+CDR of 1st arg by contents of 2nd. See page
79.

RPLACD
built in function
(RPLACD pair item)
Update the second element of a pair. See page 79 and page
88.

RPLACSTR
function, with macro definition for compilation
(RPLACSTR c-str1 s-int1 s-int2 c-str2 [s-int3 [s-int4]])
Where any of the s-int arguments may also be NIL.
RPLNODE
function
(RPLNODE pair item1 item2)
Update CAR+CDR of 1st arg by 2nd & 3rd args. See page 79.

RREAD
function
(RREAD c-str rstrm)
Read an item from a LISPLIB, by key. See page 125.

RSETCLASS
function
(RSETCLASS c-str s-int rstrm)
Sets the class byte in LISPLIB item. See page 126.

RSHUT
function
(RSHUT rstrm)
Close a LISPLIB stream. See page 126.

RWRITE
function
(RWRITE c-str item rstrm)
Write an item into a LISPLIB, by key. See page 126.

SASSOC
function
(SASSOC item list app-ob)
ASSOC with function supplied for failure. See page 86.

SBCP
function
(SBCP id)
Test for existence of a shallow binding cell

SBOUNDP
function
(SBOUNDP id)
Tests for binding in stack part of environment

SCANAND
macro
(SCANAND app-ob list ...)
Apply function to succeeding list items until NIL. See page 66.

SCANOR
macro
(SCANOR app-ob list ...)
Apply function to succeeding list items until nonNIL. See page 66.

SEARCHPAIRVECTOR
function, with macro definition for compilation
(SEARCHPAIRVECTOR item p-vec)
Searches vector for EQ item in even element

SEEWHAT
macro
(SEEWHAT id ...)
Return useful information about id's.

SEGMENTNAME
function
(SEGMENTNAME)
Returns the file name of this segment image

SEGTIME
function
(SEGTIME)
Returns time and date of segment image

SELECT
macro
(SELECT exp1 list ... exp2)
Where each list is of the form (exp exp ...). A case construct. See page 51.

SEQ
special form
(SEQ [exp ...])
Evaluate expressions sequentially, allow GOs. See page 49.

SET
built in function
(SET id item)

Table of system functions, variables and commands

177
Assign value of one argument to value of other. See page 61.

SET-ECHO-PRINT function
(SET-ECHO-PRINT boolean)
Turn echo printing on/off in SUPV

SET-GLOBAL-ID function
(SET-GLOBAL-ID id item)
Global environment assignment. See page 62.

SET-ID function
(SET-ID id item)
Sets fluid value of id in current stack. See page 61.

SET-LEX-ID function
(SET-LEX-ID id item)
Sets value of id, sees caller's lexicals. See page 62.

SET-MCASE function
(SET-MCASE str)
Forces a console stream to mixed-case reading

SET-QUAL function
(SET-QUAL str id)
Forces an input edit mode in a console stream

SET-S macro
(SET-S (id access-path instance) item)
Update a field in a defined structure.

SET-STREAM-A-LIST function
(SET-STREAM-A-LIST str list)
Update operator on fast streams. See page 124.

SET-STREAM-BUFFER function
(SET-STREAM-BUFFER str item)
Update operator on fast streams. See page 124.

SET-STREAM-P-LIST function
(SET-STREAM-P-LIST str item)
Update operator on fast streams. See page 124.

SET-UCASE function
(SET-UCASE str)
Forces a console stream to upper-case reading

SET-VALUE-PRINT function
(SET-VALUE-PRINT boolean)
Turn value printing on/off in SUPV

SETANDFILEQ macro
(SETANDFILEQ id item)
Assign and add to LISP LIB

SETDIFFERENCE function
(SETDIFFERENCE list1 list2)
Members of one list which are not in other. See page 83.

SETDIFFERENCEQ function
(SETDIFFERENCEQ list1 list2)
Same as SETDIFFERENCE, but uses EQ. See page 83.

SETELT function, with macro definition for compilation
(SETELT (vec | str | list) s-int item)

SETFUZZ function
(SETFUZZ pair)
Change the numeric comparison tolerance

SETGENLABELSEEK function
(SETGENLABELSEEK s-int)
(Re-)initializes GENLABEL seed

178 YKTLISP Program Description and Operations Manual
<table>
<thead>
<tr>
<th>Function/Keyword</th>
<th>Description</th>
</tr>
</thead>
</table>
| SETLINE         | Function
| (SETLINE s-int strm) | Set record number in FILE stream |
| SETQ            | Special form
| (SETQ id item)   | Assign item to the (unevaluated) identifier. See page 61. |
| SHARED          | Command to system interface
| (CALLBELOW 'SHARED') | Returns SHARED/NOTSHARED status |
| SHAREDITEMS     | Function
| (SHAREDITEMS item) | Return vector of multireachable items in structure |
| SHARED P        | Function
| (SHARED P)      | Tests for SHARED/NOTSHARED mode |
| SHOW-CALLS      | Macro
| (SHOW-CALLS exp) | Print a file SHOW CALLS A containing a detailed call trace. |
| SHOW-S          | Macro
| (SHOW-S id access-path instance) | Display a structure instance with field names. |
| SHUT            | Function
| (SHUT strm)     | Close a stream, non-op on console streams |
| SHUT            | Command to system interface
| (CALLBELOW 'SHUT' sysdep-area) | Close a file |
| SIN             | Function
| (SIN num)       | Sine, argument in radians. See page 111. |
| SIZE            | Function
| (SIZE item)     | Number of elements in a list or vector. See page 92, page undefined and page 90. |
| SKIP            | Function, with macro definition for compilation
| (SKIP s-int [str]) | Does n TERPRIs, may default stream. See page 122. |
| $MIN TP          | Built-in function
| ($MIN TP item)   | Test for small integer type. See page 74. |
| SORTBY          | Function
| (SORTBY app-cl list) | Sorts list of items with given access function. See page 90. |
| SORTGREATERP    | Function
| (SORTGREATERP item1 item2) | Initialized to GGREATERP, for QSORT |
| SOURCELIST      | Definition option
| (SOURCELIST , boolean) | Requests printing of the source expression on the listing stream. See page 133. |
| STACKLEFT       | Function
| (STACKLEFT)     | Returns bytes of STACK space remaining |
| STACKLIFO       | Variable
| Stream which feeds the console stack. See page 117. |

Table of system functions, variables and commands
STARTTIME
function
(STARTTIME)
Returns time/date of initial entry to LISP

STAT
command to system interface
(CALLBELOW 'STAT systemp-area)
Sends XMSG to DATASTAG

STATE
built in function
(STATE [list])
Capture and environment and control

STATEP
built in function
(STATEP item)
Test for SD type

STORECHAR
function, with macro definition for compilation
(STORECHAR c-str s-int id)
Insert character into a string. See page 101.

STRCONC
function
(STRCONC (c-str | id) ...)
Concatenate a set of strings. See page 96.

STRDEF
macro
(STRDEF structure-def)
Equivalent to LAMBDA, for structure definitions.

STREAM-A-LIST
function
(STREAM-A-LIST strm)
Returns A-list component of a fast stream. See page 124.

STREAM-BUFFER
function
(STREAM-BUFFER strm)
Returns buffer component of a fast stream. See page 124.

STREAM-DESCRIPTOR
function
(STREAM-DESCRIPTOR strm)
Returns descriptor component of a fast stream. See page 124.

STREAM-P-LIST
function
(STREAM-P-LIST strm)
Returns the system control block of a fast stream. See page 124.

STREAMP
function
(STREAMP item)
Huristic is-this-a-stream? test, = PAIRP. See page 75.

STRGREATERP
function
(STRGREATERP c-str1 c-str2)
Compare (collating sequence) two strings. See page 103.

STRINGIMAGE
function
(STRINGIMAGE item)
Convert an s-exp to a string, as if printed. See page 96.

STRINGIZE
function
(STRINGIZE item)
Makes a string of the elements of a list. See page 96.

STRINGIZER
variable
Global whose value defines the stringizer, ' 

STRINGLENGTH
function
(STRINGLENGTH str)
Number of characters or bits in a string See page 98.

STRINGP
built in function
(STRINGP item)
Test for string type. See page undefined.

STRING2BITSTRING
function

180 YKTLISP Program Description and Operations Manual
STRING2ID-N function
(STRING2ID-N c-str s-int)
Converts the nth token in a string to an id. See page 99.

STRING2PINT-N function
(STRING2PINT-N c-str s-int)
Converts the nth token in a string to a number. See page 99.

STRLENGTH function
(STRLENGTH c-str)
Synonym for STRINGLENGTH

STRPOS function
(STRPOS c-str1 c-str2 s-int item)
Searches string for sub-string. See page 99.

STRPOS function
(STRPOS table c-str s-int item)
Searches string for specified character(s). See page 99.

STRTRT function
(STRTRT table c-str pair)
Locates and identifies characters in string. See page 100.

SUBLOAD function
(SUBLOAD file (id | list))
Load a subset of a LISPLIB. See page 127.

SUBRP built-in function
(SUBRP item)
Test for SUBR type. See page 72.

SUBST function
(SUBST item1 item2 item3)
Create a structure with item substitutions

SUBSTRING function
(SUBSTRING c-str s-int1 s-int2)
Create a new string from part of another. See page 99.

SUBI function
(SUBI num)
Subtracts one from its argument. See page 108.

SUFFIX function, with macro definition for compilation
(SUFFIX id c-str)
Adds a character to a string, lengthening it. See page 101.

SUPERMAN function
(SUPERMAN)
The next-to-root function, binds error streams

SUPV function
(SUPV strm1 strm2)
System READ/EVAL/PRINT loop

SUPV-PRINT function
(SUPV-PRINT item strm)
Print function for PRINTVAL, = PRETTYPRINT

SVC202 command to system interface
(CALLBELOW 'SVC202' sysdep-area)
Issues SVC 202 for arbitrary PLIST

SYSID function
(SYSID)
Returns the cpu id for the host system
SYSID  command to system interface
      (CALLBELOW "SYSID")
      Returns 1, meaning CMS

SYSKEY  function
      (SYSKEY)
      Returns machine/time stamp for this fileim

TAB  function, with macro definition for compilation
      (TAB S-int [strm])
      Sets output position, may default stream. See page 122.

TAILP  function
      (TAILP item list)
      Test 1st arg for Eq to (C [D...]) 2nd arg). See page 85.

TEMPDEFINE  function
      (TEMPDEFINE item [list])
      Where item is either a name/expression list or a list of
      name/expression lists.
      Adds augmented STATES to OPTIONLIST

TEMPUS  command to system interface
      CALLBELOW 'TEMPUS' sysdep-area)
      Returns time, date and CPU usage

TEMPUS-FUGIT  function
      (TEMPUS-FUGIT)
      Returns elapsed virtual time

TERREAD  function, with macro definition for compilation
      (TERREAD [strm])
      Discards remainder of line, may default stream. See page 120.

TERPAGE  function
      (TERPAGE S-int [strm])
      Advance the stream to a multiple of page length.

TERPRI  function, with macro definition for compilation
      (TERPRI [strm])
      Forces output of line, may default stream. See page 121.

TEST-S  macro
      (TEST-S id access-path instance)
      Test if a defined field is present in an instance of the
      structure.

THROW  function, with macro definition for compilation
      (THROW (id | S-int) item)
      Return control to a CATCH, executing EXITs. See page 57.

THROW-PROTECT  macro
      (THROW-PROTECT exp1 exp2)
      Evaluates an expression, despite THROW/UNWIND. See page 57.

TIMES  function, with macro definition for compilation
      (TIMES num ...)
      Computes product of a set of numbers. See page undefined.

TIME2NUM  function
      (TIME2NUM c-str)
      Where c-str is a time/date, as returned from CURRENTTIME, e.g.
      Encode time-date string as small integer

TIMLL  command to system interface
      (CALLBELOW 'TIM LL')
      Returns console input line length

TOULL  command to system interface
      (CALLBELOW 'TOULL')
      Return console output line length

182  YXTLISP Program Description and Operations Manual
TPLINE
command to system interface
(CALLBELOW 'TPLINE' sysdep-area1 sysdep-area2)
Display line on console

TRACE
macro
(TRACE exp id ...)
Evaluate exp while MONITORing id .... See page 140.

TRANSLIST
definition option
(TRANSLIST : boolean)
Requests printing of the macro expanded LISP code. See page 133.

TRIMSTRING
function
(TRIMSTRING c-str)
Forces the capacity of a string to minimum. See page 101.

TRUEFN
function, with macro definition for compilation
(TRUEFN)
(LAMBDA () "T"). See page 69.

TT
macro
(TT exp)
Evaluate exp and return the total and virtual cpu time used.

TYPEOF
function, with macro definition for compilation
(TYPEOF item)
Returns the type-byte of a pointer as a SMINT

U-CASE
function
(U-CASE (c-str | id | list))
Shifts string, id, or list thereof to u case. See page 102 and page 114.

UASSOC
function
(UASSOC item list)
Search a list of name-value pairs, uses UEQUAL. See page 86.

UEQUAL
function
(UEQUAL item item2)
Tests for update equivalence. See page 76.

UMEMBER
function
(UMEMBER item list)
Searches list, using UEQUAL. See page 85.

UNEMBED
function
(UNEMBED id)
Undoes EMBED See page 142.

UNFREEZE-SHARED-SEGMENT
function
(UNFREEZE-SHARED-SEGMENT)
Reestablishes access to the shared segment

UNINTERN
function
(UNINTERN c-str)
Creates an un-interned identifier. See page 114.

UNION
function
(UNION list1 list2)
Set union of two lists. See page 83.

UNIONQ
function
(UNIONQ list1 list2)
Set union of two lists, uses EQ. See page 83.

UNSHARE
command to system interface
(CALLBELOW 'UNSHARE')
Force shared segment to non-shared mode

UNWIND
function
(UNWIND [s-int [item]])

Table of system functions, variables and commands 183
THROWS to the nth ERRSET. See page 57.

USEREXT
command to system interface
(CALLBELOW 'USEREXT' c-str [(s-int | sysdep-area)x])
Escape to locally implemented SYSDEP commands

VECP
built in function
(VECP item)
Tests for vector types. See page 73.

VECTOR
function, with macro definition for compilation
(VECTOR [item ...])
Makes a reference vector of its arguments. See page 91.

VECTOROFSAME
function
(VECTOROFSAME vec)
Tests for identical types on vector members

VEC2LIST
function
(VEC2LIST vec)
Makes a list of the contents of a vector. See page 83, page undefined vveclis, and page 99.

VERSION
function
(VERSION)
Returns string with version id, creation date

VGREATERP
function
(VGREATERP vec1 vec2)
Subroutine of GGREATERP, orders vectors

VMEMQ
function
(VMEMQ item r-vec)
Returns index of Eq element of a vector

VSCANAND
macro
(VSCANAND app-ob (vec | list) ...)
Like SCANAND, for vectors. See page 67.

VSCANOR
macro
(VSCANOR app-ob (vec | list) ...)
Like SCANOR, for vectors. See page 67.

WHO CALLED
function
(WHO CALLED s-int)
"Name" of s-intth BPI up the control chain

WHO SEES
macro
(WHO SEES id ...)
Return a list of functions that call or use the ids free or quoted.

WORDVECP
built in function
(WORDVECP item)
Tests for word vector type. See page 72.

WRAP
function
(WRAP list item)
Where item2 is either the "x" or a list of "x"s.
"Wraps" list items e.g. (a b) -> ((x a) (x b)). See page 68.

WRBLK
command to system interface
(CALLBELOW 'WRBLK' s-int1 s-int2 s-int3
sysdep-area1 sysdep-area2)
Write blocked records to a file

WRITE
function, with macro definition for compilation
(WRITE character strm)
Places a character into an output stream. See page 121.

WRITE-LINE
function
(WRITE-LINE c-str strm)
Writes a line on an output stream. See page 121.

184 YKTLISP Program Description and Operations Manual
WSI}ME function
(WSI}ME)
Returns time and date of WSI creation

XORBIT function
(XORBIT b-str ...)
Exclusive OR of bit strings. See page 97.

XORCAR function, with macro definition for compilation
(XORCAR item)
If arg is pair, CAR, otherwise arg. See page 78.

XORCDR function, with macro definition for compilation
(XORCDR item)
If arg is pair, CDR, otherwise arg. See page 78.

ZERO} function
(ZERO} item)
Tests for zero value. See page 105.

32XXPLST command to system interface
(CALLBELOW '32XXPLST' s-int sysdep-area)
Returns information about a virtual device

32XXWRIT command to system interface
(CALLBELOW '32XXWRIT' s-int1 sysdep-area s-int2 s-int3)
Performs DIAG 55 output to console
Special Characters

...Q functions 45
.ANOVAL 145
& 17, 135, 145
$AILFILES 145
$CHECKMODE 145
$CLEAR 145
$ERASE 145
$EST 145
$FCOPY 145
$FILEDATE 145
$FILEREC 145
$FILESIZE 145
$FINDLINK 145
$FLAT-STRING 145
$FMFTM 145
$INLINE 145
$INLINE 145
$MEMORY 146
$MOVEFILE 146
$OUTFILE 146
$OUTSTREAM 146
$REPLACE 146
$RESET-STREAM 146
$SCREENSIZE 146
$SETPFIMM 146
$SHOWLINE 146
$T-DELTA 146
$T-TIME 146
$TIMESTAMP 146
$QUOTE 146
$TYPELINE 146
$V-DELTA 146
$V-TIME 146
$CODE 146
\ in names 45
? 147

INDEX

Batcherror 148
BITERROR 103, 148
BITSTRINGP 72, 148
BITSTRINGSTRING 96, 148
BITSTRING2NUM 148
BOOLEYNP 148
BOUNDEDBY? 148
BOUNDP 148
BPFILET 148
BPFILET 133, 148
BPROPNAME 149
BPSTP 73, 149
BREAK 149
break loop 17
built in function application 39
BYTES 149

C

CHR 149
C(A|D) 77
CAID... .R 84
CAAAAR 77, 149
CAADR 77, 149
CAAR 77, 149
CAADR 77, 149
CAADDR 77, 149
CADDR 77, 149
CADDR 77, 149
CADDR 77, 84, 149
CADDR 77, 84, 149
CADDR 77, 84, 149
CALL 150
CALLBELOW 150
CALLEDBY? 150
CALLX 150
CAR 77, 84, 124, 150
CASEGO 50, 150
CATCH 41, 55, 150
CBOUND 150
CD... .R 84
CDAAR 77, 150
CDAADR 77, 150
CDAAR 77, 150
CDAADR 77, 150
CDAADR 77, 150
CDAAR 77, 150
CDAAR 77, 150
CDAADR 77, 150
CDAAR 77, 150
CDAAR 77, 150
CDAADR 77, 150
CDAADR 77, 84, 151
CDAADR 77, 84, 151
CDAAR 77, 84, 151
CDAAR 77, 84, 151
CEVAL-ID 60, 151
CEVAL-LX-ID 61, 151
CHANGELENGTH 101, 151
CHARP 73, 151
CHAR2NUM 151
CLOSEDFN 54, 151
CLOSURE 151
CONC 82, 151
comma names 45
COMMENT 151
compilation, environment. 42
COMPILE 131, 151
COMP370 151
CONS 77, 81, 152
CONSOLEPRINT 123, 152
CONSTANT 47, 152
CONTAINSQ 152
control chain 42
control of execution. 47-58
operators 47-58
conditional evaluation 50-52
multiple level returns 55-58
operator specification,
  binding 55-56
sequence of evaluation 48-50
value specification 47-48
CONVERSATIONAL 152
CONVVAL 152
COPY 152
COS 111, 152
COUNT 152
COUNT-CALLS 152
COUNT-PAIRS 152
CREATE-BSC 152
CSET-ID 62, 152
CSET-LEX-ID 62, 152
CURINSTREAM 117, 152
CURLINE 152
CUROUTSTREAM 117, 153
CURRENTTIME 153
CURRINDEX 153
CYCLES 153
CYCLES-P 153

DCQ 153
debugging 17
debugging, operators which
  aid 135-143
  stack examination 135-140
  tracing execution 140-143
DEBUGMODE 153
DEFINE 129, 153
DEFINE 151, 153
DEFINE-STRUCTURE 153
definition, operator 129-134
  definition 129-131
  definition options 131-134
DEFINSTREAM 117, 153
DEFLIST 115, 153
DELPROP 153
DEPLAD 128, 153
DIFFERENCE 108, 153
DIGIT 74, 153
DIGIT-FIX 154
DISABLE 154
discontiguous shared segment 7, 8,
  10

DISPATCHER 154
DIVIDE 109, 154
DROPAREA 154

E

EBCDIC 154
ECONST 154
ECQ 154
EDIT 154
EFFACE 89, 154
EFFACEQ 90, 154
ELT 84, 93, 98, 154
EMBED 143, 154
EMBEDDED 143, 154
empty-stream 119
ENABLE 154
ENBLSPIE 154
end-of-file 119
ENVIRON 154
ENVIRONAL 154
environment chain 42
environment of compilation. 42
environment operators on 59-62
assignment 61-62
evaluation 59-61
EOF 154
EOLP 155
EQ 75, 155
EQSUBLIST 155
EQSUBSTVEC 155
EQUAL 76, 155
EQUAL/EQ use 45
EQUALN 76, 155
ERASE 155
ERRCATCH 58, 155
ERROR 155
error handling 17
error message 17
ERROR-PRINT 155
ERRINSTREAM 117, 155
ERRN 155
ERROUTSTREAM 117, 155
ERRRR 155
ERROR 2 155
ERROR 3 155
ERSET 57, 155
ERR 155
ERR 2 155
ERR 3 155
ERR 5 155
ERR 6 155
ERR 7 155
EVAL 42, 59, 156
EVAL-supervisor 17
EVAL-GLOBAL-ID 60, 156
EVAL-ID 59, 156
EVAL-LEX-ID 60, 156
EVALANDFILEACTQ 156
EVAL 59, 156
EVALFUN 59, 156
EXF 156
EXIT 49, 156
EXP 111, 156
EXPT 111, 156
EXTERNAL-EVENTS-CHANNELS 156
EXITSTATE 156

188  YKLISP Program Description and Operations Manual
INDEX

F

F 157
Fast streams 123
FASTSTREAM 75, 157
FETCH 157
FETCHCHAR 98, 157
GETCHPROP 157
GETCHPSMINT 157
FILE 134, 157
FILEACTQ 157
FILEIN 157
FILELISP 6, 15, 157
FILEOUT 157
FILEQ 157
FILESEG 157
FIN 5, 17, 157
FIX 105, 157
FIXP 74, 157
FLOAT 105, 157
FLOATP 74, 157
FLUID 41
FORCE-GLOBAL 157
FRMCODE 158
FRARGCOUNT 158
FREEZE-SHARED-SEGMENT 158
FRP 73, 158
funarg 42, 55, 158
funarg application 40
FUNARGP 75, 158
FUNCTION 55, 158
functional application 39
fuzz 29

G

GCACCOUNT 158
GCMSG 158
GCNPLIST 158
GENLABEL 114, 158
GENSYM 114, 158
GENSYMP 73, 158
genyms 113
GET 86, 119, 158
GETBITSTR 95, 158
GETCH 158
GETFLT 158
GETFULLSTR 95, 159
GETHASLABRARY 159
GETHAXSTRING 159
GETREALV 91, 159
GETREFV 91, 159
GETSTR 95, 159
GETWORDV 91, 159
GETZEROVEC 91, 159
GFIPLIST 159
GGREATERP 159
global environment 41
GLOBAL VALUES
  .NOVAL 145
  CURINSTREAM 117, 152
  FAUROUTSTREAM 117, 153
  ERRORINSTREAM 117, 155
  ERROROUTSTREAM 117, 155
  EXTERNAL-EVENTS-CHANNELS 156
  F 157
  INITSUPV 160
  lettrize 26, 120, 161
  MONITOR-ON-OFF 165
  NULLOUTSTREAM 117, 166
  OPTIONLIST 167
  PROGRAM-EVENTS 169
  QUOTEIZER 120, 174
  RESTART 176
  STACK 17
  STACKLIFO 117, 179
  STRINGIZER 120, 180
  global variable description 43
  GLODESTATE 159
  GO 49, 159
  GREATERP 106, 159

H

HASHARRAYP 159
HASHHIT 159
HASHITEM 159
HASHPROP 159
HEAPLEFT 160
HEXECP 160
HEXNUM 160
HEXSTRINGPART 160
HPROPLIST 160

I

identifiers, operations on 113-116
accessing 114-115
accessing the object array 116
creation 113-114
searching and updating 115-116
updating 116
IDENTITY 69, 160
IDENTP 73, 160
IFCAR 78, 160
IFCDR 78, 160
INITIALOPEN 160
INITSUPV 160
INITSYMTAB 133, 160
INTERN 113, 160
INTERSECTION 83, 160
INTERSECTIONQ 83, 160
IOSTATE 160
IOSTATEW 160
IS-CONSOL 75, 160
ITEM-N-ADV 119, 161
iteration, operators 63-69
auxiliary operators 69
iteration over vectors 66-68
iterations over lists 63-66
miscellaneous iteration
operators 68-69

J

jaunt 40

K

key word description 43

Index 189
| L | L-CASE 102, 114, 161  
   LAM 161  
   LAMBDA 41, 52, 161  
   LAPLST 133, 161  
   LAST 84, 161  
   LAST-EXP 17, 161  
   LAST-VALUE 17, 161  
   LASTNODE 84, 161  
   LEFTSHIFT 110, 161  
   LENGTH 90, 161  
   LENGTHCODE 93, 98, 161  
   LESSP 106, 161  
   letterizer 24, 161  
   libraries, operations on 125-128  
     creation 125  
     data input 125-126  
     data output 126  
     library management 126-127  
     program loading 127-128  
   LINTP 74, 161  
   LISP CMS MODULE 5  
   LISPRETX 161  
   LISPSEQ 101  
   LIST 81, 161  
   LIST-PAIRS 161  
   LISTING 134, 162  
   LISTOFFFLUIDS 162  
   LISTOFFFREE 162  
   LISTOFFUNCTIONS 162  
   LISTOFFLEXICALS 162  
   LISTOFFQUOTES 162  
   LISTOFFSAME 162  
   LISP 72, 162  
   lists, operations on 81-90  
     accessing 84-85  
     creation 81-84  
     miscellaneous 90  
     searching 85-87  
     searching and updating 87  
     updating 87-90  
   LIST2FLITEC 91, 162  
   LIST2VEC 92, 162  
   LEX 111, 162  
   LOADCOND 127, 162  
   LOADVOL 127, 162  
   LOG 111, 162  
   LOG2 111, 162  
   LOTSOF 81, 162  

| M | MAADDTEMPDEFS 132, 163  
   macro application 40  
   macro definition for compilation 39  
   MACRO-APP-SD 132, 163  
   MAKEPROP 87, 115, 163  
   MAKESTRING 97, 163  
   MAKESTRING 97, 163  
   MAP 68, 163  
   MAPCAR 68, 163  
   MAPE 67, 163  
   MAPLE 67, 163  
   MAPLIST 68, 163  
   MAPOBITS 68, 163  
   MAPSET 67, 163  
   MASKNUM 163  
   MATEMDEFINE 132, 163  
   MATEMSETQ 163  
   MAX 106, 163  
   MAXINDEX 92, 163  
   MDEF 42, 164  
   MDEFX 164  
   MDO 164  
   MEMBER 85, 164  
   MEMQ 85, 164  
   MESSAGE 134, 164  
   MIN 107, 164  
   MINUS 107, 164  
   MINUSP 106, 164  
   MLAMBDA 53, 164  
   MMAP 64, 164  
   MMAPC 64, 164  
   MMAPCAN 65, 164  
   MMAPCAR 64, 164  
   MMAPCON 65, 164  
   MMAPLACA 65, 164  
   MMPALT 64, 165  
   MONITOR 161, 165  
   MONITOR-ON-OFF 165  
   MOV2VEC 94, 165  
   MRP 73, 165  
   MSUBRP 72, 165  
   MTON 83, 165  
   NAMEDERRSET 58, 165  
   NCORNC 83, 165  
   NFCORNC 165  
   NEWAREA 165  
   NEWQUEUE 165  
   NEXT 119, 165  
   NILFN 69, 165  
   NILLEFT 165  
   NILSD 165  
   NOLINK 132, 165  
   NONINTERREXIBLE 133, 165  
   NONSTOREDP 71, 166  
   NOT 71, 166  
   NOTFILE 166  
   NREVORE 89, 166  
   NSTACKED 166  
   NSUBST 166  
   NTUPLEP 166  
   nucleus extension 8, 13  
   NULL 71, 166  
   null line 17  
   NULLOUTSTREAM 117, 166  
   NUMBEROFAROS 166  
   NUMBERP 74, 166  
   numbers, operations on 105-111  
     computation 106-111  
     conversion 105  
     predicates 103-106  
   numeric comparisons 29  
   NUM2TIME 166  

190 YKLISP Program Description and Operations Manual
OBARRAY 116, 166
OBDUMP 166
OBEY 166
ODDP 106, 166
ONE-OF 166
OP-RECOGNITION-5D 132, 167
operations on identifiers 113-116
accessing 114-115
accessing the object array 116
creation 113-114
searching and updating 115-116
updating 116
operations on libraries 125-128
creation 125
data input 125-126
data output 126
library management 126-127
program loading 127-128
operations on lists 81-90
accessing 84-85
creation 61-84
miscellaneous 90
searching 85-87
searching and updating 87
updating 87-90
operations on numbers 105-111
computation 106-111
conversion 105
predicates 105-106
operations on pairs 77-79
accessing 77-78
creation 77
updating 78-79
operations on streams 117-124
accessing components 123-124
creation 117-119
data input 119-121
data output 121-123
updating components 124
operations on strings 95-103
accessing 98-99
comparing 103
creation 95-98
searching 99-100
updating 101-103
operations on vectors 91-94
accessing 92-94
creation 91-92
updating 94
operations which act as
predicates 71-76
comparing 103
other 75-76
type testing 71-75
operator definition 129-134
definition 129-131
definition options 131-134
operators on the environment 59-62
assignment 61-62
evaluation 59-61
operators which aid
debugging 133-143
stack examination 135-140
tracing execution 140-143
operators which control
evaluation 47-58
conditional evaluation 50-52
multiple level returns 55-58
operator specification,
binding 52-55
sequence of evaluation 48-50
value specification 47-48
operators which iterate 63-69
auxiliary operators 69
iteration over vectors 66-68
iterations over lists 63-66
miscellaneous iteration
operators 68-69
OPTIMIZE 134, 167
OPTIONLIST 167
OR 51, 167
ORADDDMPEPS 131, 167
ORBIT 97, 167
ORTEMPEDEFINE 152, 167
ORTEMPSETQ 167

PACKHEXSTRING 167
PAIRP 72, 167
pairs, operations on 77-79
accessing 77-78
creation 77
updating 78-79
PAVIICMSG 167
PARAMETERS 167
PARMLIST 167
PLACEP 74, 167
PLEXP 167
PLUS 107, 167
PLUSP 105, 167
pname 24, 97, 114, 168
POINFILE 168
POLLP 168
POP 168
POP 168
POST 168
POST-SELECT 168
predicates, operations which act
as 71-76
comparing 103
other 75-76
type testing 71-75
PRETTYPRINT 123, 168
PRETTYPRINT 123, 168
PRINTM 123, 168
PRINT 123, 168
PRINTCH 121, 168
PRINTEXP 121, 168
PRINTIVAL 168
PRINTWARM 168
PRINO 122, 168
PRINOR 168
PRINI 122, 168
PRINIB 122, 169
PRGO 49, 169
PROGN 48, 169
PROGRAM-EVENTS 169
PROGI 48, 169
PROGO 48, 169
PROPLIST 113, 169
PRY 169
PSMINTP 169
PUSH 169
PUTBACK 121, 169
Index 193

RREAD 125, 177
RSETCLASS 125, 177
RSHUT 126, 177
WRITE 126, 177

SASSOC 86, 177
SBCP 177
SBORDERP 177
SCHAND 66, 177
SCANOR 66, 177
SEARCHPAIREDVECTOR 177
SEESWHAT 177
SEGMENT 5
SEGMENTNAME 177
SEGTIME 177
SELECT 51, 177
SEQ 49, 177
SET 61, 177
SET-echo-print 17, 178
SET-GLOBAL-ID 62, 178
SET-ID 61, 178
SET-LEX-ID 124, 178
SET-MCASE 178
SET-QUAL 178
SET-S 178
SET-STREAM-A-LIST 124, 178
SET-STREAM-BUFFER 124, 178
SET-STREAM-P-LIST 124, 178
SET-UCASE 178
SET-VALUE-PRINT 17, 178
SETANDFILEQ 178
SETDIFFERENCE 83, 178
SETDIFFERENCEQ 83, 178
SETELY 88, 94, 101, 178
SETFUZZ 178
SETGENLABELSEED 178
SETLINE 179
SETP 61, 179
shallow binding call 42
SHARED 179
SHAREDITEMS 179
SHAREDP 179
SHLSPWS 5, 6
SHOW-CALLS 179
SHOW-S 179
SHUT 118, 179
SIN 111, 179
SIZE 90, 92, 179
SKIP 122, 179
smallest-integer arithmetic 45
SMINTP 74, 179
SORTBY 90, 179
SORTGREATERP 179
SOURCELIST 133, 179
special form application 39
STACKLEFT 179
STACKLIFO 179
STACKLIFO variable 117
STARTIME 180
STATE 180
STATE descriptor 42
state descriptor application 40
STATEP 75, 180
storage organization 7
BPISEC 9
diagram 10
FIXEDSEC 9
HEAP 9
loader options

BPI 9
CMHIGH 9, 10
COMMAND 9
default values 9
FIXED 9
format 9
GETMIN 9, 10
HEAP 9
KEY 9
NIL 9
NONSHARE 9
SHARE 9
STACK 9
SYSTOR 7, 9
USERSTOR 7, 9
NILSEC 9
STACK 9
storage protection 7, 9
STORECHAR 101, 180
STROCONC 96, 180
STRODF 180
STREAM-A-LIST 124, 180
STREAM-BUFFER 124, 180
STREAM-DESCRIPTOR 124, 180
STREAM-P-LIST 124, 180
STREAM 75, 180
streams, operations on 117-124
accessing components 123-124
creation 117-119
data input 119-121
data output 121-123
updating components 124
STRINGGRATERP 103, 180
STRINGIMAGE 96, 180
STRINGIZE 96, 180
STRINGIZER 180
STRINGLENGTH 98, 180
STRINGP 72, 180
strings, operations on 95-103
accessing 98-99
comparing 103
creation 95-98
searching 99-100
updating 101-103
STRING8BITSTRING 96, 180
STRINGID-N 99, 181
STRING2PRINT-N 99, 181
STRINGLENGTH 181
STRINGPOS 93, 181
STRINGPSL 100, 181
STRINGP 100, 181
structure access 40
structured by 52
SUBLOAD 127, 181
SUBRP 72, 181
SUBST 181
SUBSTRING 99, 181
SUB1 108, 181
SUFFIX 101, 181
SUPERMAN 181
SUPV 17, 181
SUPV-PRINT 181
SVC202 181
SYSID 181, 182
SYSKEY 182
SYSTBUILD COMMANDS (CALLBELOWs)
? 147
COWSAL 152
DROPREA 154
ENBLSPIE 154
ERASE 155
FILEIN 157
FILEOUT 157
GCMPIST 158
GFIPLIST 159
ZEROP 105, 185

32XXPLST 185
32XXWRIT 185