CHAPTER 9. HIGH LEVEL UTILITIES

LA.HI file contains the utuilities which extends LaForth to a system useful for program developments. Most words are defined in a single line. Without stack pictures and some comments, it is very difficult to understand what the words are doing. Although the glossary is supplied in LA.DEF and LA.SUM files, it still takes considerable efforts to put the pieces together to get a whole picture. In this chapter we try to arrange the words and their descriptions together, with related words grouped into sections. Having the definition and the description of a word in the same place makes it easier to grasp the real contents.

9.1. BASIC TOOLS

\ –
Begin a comment. The comment is terminated by another \ or the end of a line.
: \ #0D -WORD DROP ; IMM
\ 23 Mar 1987 VERSION.

(–
Begin a comment. The comment is terminated by another (, or the end of a line.
: (#29 -WORD DROP ; IMM

." --
Print the string which follows. A " mark followed by a space, tab, or carriage return terminates the string. The string can be any length and contain any 8-bit ASCII character. If an imbedded " mark is desired, it can be followed by any printable non-blank character, or alternatively each pair of "" characters may be used to represent a single " mark. Imbedded Line-Feeds and carriage returns are allowed. Note, however, that a Line-Feed does not automatically follow a Return--even though on input Return does echo the Line-Feed.
: ." ?COMP COMPILE (." HERE #22 TEXT! 1+ DP ! ; IMM

PAD – addr
The address of a temporary storage area, generally placed at SO + 2.
: PAD (-- a) SO 2+ ;

" -- address
Begins a quoted string. This is a state-smart word which may be use either inside or outside of colon definitions. The string is terminated by another " mark and a following space, tab, or carriage return. The count of characters within the string is found at the word located at "address-2". In addition, the string is terminated by an ASCII null, not included in the string count. If the string is used outside of a colon definition, it is transitory, and will be over-written by other words which use PAD, just above the stack. A " mark may be placed in the text string by using two of them, "" . A " mark may also be followed within the string by any other printable character other than a space.

```
: "     STATE @ ?[ COMPILE (" HERE 0 , HERE DUP #22 TEXT!
                DUP 1+ DP ! SWAP - SWAP !
            ][ PAD #22 TEXT! DROP PAD          ]? ; IMM
```

HEX –
Sets the value of BASE to sixteen.
: HEX (--) $16 BASE ! ;

DEPTH -- n
Return the number of items on the data stack.
: DEPTH (-- n) SP@ SO SWAP - 2/ ;

9.2. CONTROL STRUCTURES

-IMM –
Compile the next word in the input stream. This is used to force the compilation of immediate words, which are normally executed in a colon definition.
```
: -IMM    ' , ; IMM
```

[[–
Mark position where ?] or]] will go back to. Alternate use is to mark the beginning of a series of conditionals like ?[or =?[and terminated by]]? .
```
: [[      ?COMP HERE 0 ; IMM
```

]] –
Unconditional jump back to [[. It also branches back to [and marks where ?[will go when the proceeding ?[finds top 'false'
```
: NRES    [[ DUP ?[ -IMM ]? 0 ][ -1 ]? ?] DROP ;
: ]]      ?COMP COMPILE 0 0 , NRES -2 ALLOT , ; IMM
```

]: –
Resume suspended : definition. The stack must be as it was when ;[was executed.

]? –
Demarks end of ?[logic.
```
: ]?      ?COMP HERE SWAP ! ; IMM
```

][–
Otherwise. Separates the 'true' and 'false' logic in conditional.
```
: ][      ?COMP COMPILE 0 HERE 0 , SWAP -IMM ]? ; IMM
```

]]? –
Terminate a series of conditionals. Used in the sequence like:
```
                [[ ... ?[ ... ?[ ... ]]?                    or:
                [[ ... =?[ ... ][ ... =?[ ... ][ ... ]]?
: ]]?     NRES DROP ; IMM
```

;[–
Suspend : definition temporarily. When]: is subsequently executed the stack must be as it was when ;[was used.
```
: ;[      ]: 0 STATE ! ; IMM
```

?[n --
Test. Do following if value on top is true, otherwise skip to matching]? or][or]]
```
: ?[      ?COMP COMPILE (?] HERE 0 , ; IMM
```

?] n --
Jump back to [[if value on top is zero.
```
: ?]      ?COMP COMPILE (?] DROP , ; IMM
```

=?[n1 n2 -- or n1 n2 -- n1
Start a CASE branch clause. When executed within a colon definition, the top two stack elements ar tested for equality. If the top two elements have a different value, the top element is dropped and the next in-line value is taken as the branch address. If the two elements are equal, both elements are dropped, the in-line value is skipped, and execution continues.
```
: =?[     ?COMP COMPILE (=?[ HERE 0 , ; IMM
```

]# –

End of indexed loop. If the index value at the top of the R-stack is non-zero, decrement the index value by one, and loop back to the start of the loop (just after the #[ ). If the index is zero, pop the value from the R-stack, and exit from the loop. Must be used within a colon definition.

```
: ]#     ?COMP COMPILE (]# , ; IMM
```

#[n --

Marks start of a decrementing indexed loop. Initial index value is moved to the R-stack. The end is marked by]# . Must be used within a colon definition.

```
: #[     ?COMP COMPILE >R HERE ; IMM
```

9.3. MATH EXTENSIONS

Many math operations can be derived simply from those in the kernel. Many of then are put in the LA.HI file because they need the high level control structures. The means of them are self evident and no comments are required here.

```
: /      ( n1 n2 -- quot )  /MOD NIP ;
: MOD    ( n1 n2 -- mod )  /MOD DROP ;
: */MOD  ( n1 n2 n3 -- mod quot )  >R M* R> UM/MOD ;
: */     ( n1 n2 n3 -- quot )  */MOD NIP ;
: =      ( n1 n2 -- flag )  - 0= ;
: >      ( n1 n2 -- flag )  SWAP < ;
: ABS    ( n1 -- n2 )  DUP 0< ?[ NEG ]? ;
: DABS   ( d1 -- d2 )  DUP 0< ?[ DNEG ]? ;
: MAX    ( n1 n2 -- n3 )  2DUP < ?[ SWAP ]? DROP ;
: MIN    ( n1 n2 -- n3 )  2DUP > ?[ SWAP ]? DROP ;
: OCTAL  $8 BASE ! ;
: BINARY  $2 BASE ! ;
: ESC    27 EMIT ;
```

9.4. NUMBER OUTPUT WORDS

SIGN n -- n

Outputs a dash if the value on top is negative.

```
: .SIGN  ( n -- n )  DUP 0< ?[ ." -" ]? ;
```

<# n1 n2 -- 0 n1 n2

Insert zero under top word on stack. Used to begin numeric output of a double precision number.

```
: <#     ( n1 n2 -- 0 n1 n2 )  0 -ROT ;
```

lo hi -- char lo' hi'

Divide double number on top by value in BASE, leave digit as an ASCII character, with double quotient on top.

```
: #      ( ud1 -- ch ud2 )  0 BASE @ UM/MOD >R BASE @ UM/MOD >R
           DUP #9 > ?[ #7 + ]? #30 + R> R> ;
```

#>. ? 0 0 --

Discard final quotient (value 0.0), then output ASCII digits from stack until NULL character is reached.

```
: #>.    ( 0 ch ch ... ch 0 0 -- )  DROP DROP [[ ?DUP ?[ EMIT ]] ;
```

#R dbl -- ?

Convert double top to ASCII digits using # and output enough spaces to right justify when digits are printed.

```
: #R     ( ud n -- )  1- 0 MAX #[ OVER OVER OR ?[ # ][ SPACE ]? ]# ;
```

#S dbl -- ?

convert double top to ASCII digits (one digit per word on stack) using # . Leaves quotient value of zero on top. This is useful when it is necessary to force leading zero digits.

```
: #S     ( ud -- ch ch ... ch 0 0 )  # [[ OVER OVER OR ?[ #  ]] ;
```

.R n1 n2 --

Top specifies the width of the field, 2nd is unsigned and printed with a space following, BASE determines the conversion.

```
: .R     ( u n -- )  >R 0 <# # R> #R #>. SPACE ;
```

.D$ d cnt --

Display the signed double precision number in a field cnt characters wide. If the field is wider than necessary, leading spaces will be displayed. If the field is too narrow, characters may be truncated. If the double number is negative, the count must include a place for a leading minus sign. The count must also include room for a "$" sign, an imbedded ".", and two numbers to the right of the decimal point.

```
: .D$    ( d n -- )  OVER 0< ?[ 1- ." -" ]?
        >R DABS <# # # #2E -ROT # R> 5 - #R ." $" #>. ;
```

.$ n1 cnt --

Print the value n1 in a field cnt characters wide. The count includes a possible leading minus sign, a "$" sign, and an imbedded decimal point, and two digits to the right of the decimal point.

```
: .$     ( n1 n2 -- )  OVER 0< SWAP .D$ ;
```

UD.R ud n --

Print an unsigned double integer in an n-column field.

```
: UD.R    >R <# # R> #R #>. SPACE ;
```

UD. ud --

Print an unsigned double integer in a free format; i.e., followed by a space.

```
: UD.     <# #S #>. SPACE ;
```

U. un --

Print an unsigned integer.

```
: U.     0 UD. ;
```

. n --

Print the signed value of top followed by one space. Numeric conversion determined by value in BASE.

```
: .      .SIGN ABS U. ;
```

D. d --

Print a signed double integer, followed by a space.

```
: D.     .SIGN DABS UD. ;
```

SPACES n --

Output n spaces to the terminal.

```
: SPACES  1- #[ SPACE ]# ;
```

.A n --

Displays the low order 8 bits from top as an ASCII character. Control characters are shown as ^ followed by the character, other characters are shown as a space then the character. This is useful to make the control characters displayable.

```
: .A     #7F AND DUP #20 < ?[ ." ^" #40 + ][ SPACE ]? EMIT ;
```

9.5. DICTIONARY WORDS

PRE addr1 -- addr2 addr1
Top must contain a word address. PRE puts the address of the preceding dictionary word under it. PRE calls QUIT when at the end of the dictionary. Used in the dictionary printing words.
```
: PRE    'PRE SWAP ?DUP 0= ?[ QUIT ]? ;
```

.DICT --
Print out he names of words defined in the dictionary.
```
: .DICT  0 VOCTABLE @ 3 +
       [[ CR 6 #[  PRE DUP .NAME 3 - $9 #[ DUP C@ ?[ 1- ][ SPACE ]? ]#
       DROP ]# PRE .NAME KIN ]] ; -2 ALLOT
```

DICT --
Prints out the location, in hex, and the names in the SEARCH vocabulary, from last defined down. DICT prints slow for ease of reading and to be stoppable, it uses SCR.
```
: DICT   0 VOCTABLE @ 3 + [[ CR PRE .W .NAME KIN ]] ; -2 ALLOT
```

~ --
Terminate a colon definition, if necessary, then delete the last dictionary entry from the GROWING vocabulary. Finally, print the name of the dictionary entry which is now the most recent.
```
: ~      STATE @ ?[ -IMM ; ]? DEFS 'LAST CHOP 'LAST .NAME SPACE ; IMM
```

CHOP addr --
Top must contain the address of a dictionary word. CHOP deletes all words back through that word from the dictionary.
```
: CHOP1  [[ 1+ @ OVER OVER U< 0= ?] ;
: CHOP2  2+ [[ 1- DUP C@ #7F AND 0= ?] DP ! ;
: CHOP   4 - $31 #[ VOCTABLE 2I + 1- CHOP1 VOCTABLE 2I + ! ]#
       CHOP2  VOCTABLE $62 + @
       $30 #[ VOCTABLE 2I + @ OVER U< 0= ?[ DROP VOCTABLE 2I + @ ]? ]#
       LATEST ! ;
```

FORGET --
Forget from the dictionary the word specified by the next string, and all the words which were defined after it.
```
: FORGET  DEFS ' CHOP ;
```

.VOCAB addr --
Print the vocabulary whose pointer address is on top. Typical use is: SEARCHING .VOCAB
```
: .VOCAB  @ 3 - .NAME SPACE ;
```

UNLINK addr1 -- addr1 addr2
Unlink the word whose address is on the stack from the vocabulary thread it belongs. addr2 is the address of the word prior to the unlinked word in the same thread. It is used as: ' CON UNLINK
```
: UNLINK  >R 0 'LAST [[ NIP 'PRE DUP I = ?] 2- @ SWAP 2- ! R> ;
```

LINK addr1 addr2 --
Link the word at addr2 to follow the word at addr1 in a vocabulary thread. UNLINK and LINK allow the user to rearrange the vocabulary threads.
```
: LINK   2- SWAP 2- OVER @ OVER ! 1- SWAP ! ;
```

ONLY --
Install ROOT as the only vocabulary on the vocabulary stack.
```
: ONLY   SEARCHING $16 - $16 0 FILL -IMM ROOT ;
```

ALSO --

Push the SEARCHING vocabulary on the vocabulary stack, making the current SEARCHING vocabulary always available for word searching.
```
: ALSO    SEARCHING $12 - DUP 2- $14 CMOVE ;
```

PREVIOUS –
Remove the top vocabulary from the vocabulary stack. Opposite of ALSO.
```
: PREVIOUS    SEARCHING $16 - DUP 2+ $16 CMOVE SEARCHING @ 3 - INSTALL ;
```

9.6. INTERACTIVE WORDS

FORTH –
This is an infinite loop of INTERPRET. It is used to invoke the entire FORTH system from within a word. Executing the word 'back-space' will return control to the word which used the word FORTH.
```
: FORTH   [[ INTERPRET ]] ; -2 ALLOT
```

KIN --
Executes FORTH if there are any characters in the input buffer. Used to provide the ability to stop when a character is received from the keyboard. The time it delays is determined by the count value in RATE.
```
USER @ :CON RATE 2 USER +!
: KIN    RATE #[ INCNT ?[ FORTH ]? ]# ;
```

:BUF –
Defining word which creates a CONSTANT whose value is the beginning of a new buffer. Also terminates the last buffer.
```
: :BUF    0 LT DROP TP @ XC!  1 TP +!  TP @ :CON ;
```

TEXT –
Accept characters from the input stream and leave them in the text buffer. The input stream is terminated by a CTRL Z character.
```
: TEXT    #1A (TEXT ;
```

PROG --
Enters text into the text buffer as does TEXT, but this then automatically RUNs the text which was just entered. This is useful in the development of programs, because it captures in the text buffer the source of each definition. See REDO.
```
: PROG    TEXT LT RUN ;
```

REDO –
This empties the last entered text in the TEXT buffer, and then does a PROG. Useful with PROG in capturing the source of new word definitions.
```
: REDO    LT MT PROG ;
```

NUM -- n
This requests a number from the keyboard and puts it on top.
```
: NUM    [[ #20 -WORD DROP (NUM 0< 0= ?] ;
```

9.7. MEMORY DUMP

DUMP addr1 addr2 –

Display the memory contents from addr1 to addr2 in a conventional core dump format. Addresses and contents are displayed in HEX, with an ASCII dump on the right.
```
: DUMP   ^ #FFF0 AND [[ CR PH #10 - PASC KIN 2DUP U< ?] 2DROP CR ;
```

```
^              u1 u2 -- u1 u2  or  u1 u2 -- u2 u1
```
SWAP top tow unsigned numbers if top is less than second.
```
: ^     2DUP U< ?[ SWAP ]? ;
```

```
PH             addr – addr+16
```
Dump 16 bytes starting at addr1.
```
: PH    .W SPACE $15 #[ DUP C@ .B DROP 1+ ]# ;
```

```
-PRNT?         char –
```
Replace a control character by ~, ASCII #7F.
```
: -PRNT?  DUP #20 < OVER #7F > OR ;
```

```
PASC           addr – addr+16
```
Dump 16 bytes from addr in ASCII form.
```
: PASC    SPACE SPACE $15 #[ DUP C@ -PRNT? ?[ DROP #7E ]? EMIT 1+ ]# ;
```

9.8. MISCELLANEOUS WORDS

```
NEW                        n --
```
Puts the value in top into the constant whose name follows. This is the safe way to change the value of a :CON
```
: NEW    ' 1+ DUP @ OVER + 2+ LIT @ = ?[ 2+ ! ][ ?COMP ]? ;
```

```
EXC                        addr1 addr2 --
```
Exchanges the values of the two words whose addresses are in top two words on stack.
```
: EXC    2DUP @ SWAP @ ROT ! SWAP ! ;
```

```
GET                        n -- val
```
Obtain the n-th item from the stack and push it on top. Note that 0 GET is equivalent to DUP .
```
: GET    1+ 2* SP@ + @ ;
```

```
PUT            val n --
```
Store "val" at the n-th item on the stack, losing the previous contents. Note that -1 0 PUT will replace the top item with a value of -1 .
```
: PUT    2+ 2* SP@ + ! ;
```

```
SCNT           -- n
```
Pushes count of words on the stack, not counting itself.
```
: SCNT    SP@ SO SWAP - 2/ ;
```

```
..             ... --
```
Print all the numbers on the data stack and clear the data stack.
```
: ..     SCNT 1- 0 MAX #[ . ]# ;
```

```
S=             –
```
Display all the numbers on the data stack without disturbing them.
```
: S=     SCNT 1- DUP 0< ?[ DROP ." Empty Stack " ][ #[ I GET U. ]# ]? ;
```

```
VU             –
```
Run the text interpreter while showing the contents of the data stack. This is very useful for teaching and debugging.
```
: VU     [[ INTERPRET CR $12 SPACES ." S=" S= #D EMIT ]] ; -2 ALLOT
```

```
PRIME       -
Sieve of Erostothenes. A standard benchmark program for integer operations.
8190 :CON SIZE  0 :VAR FLAGS SIZE ALLOT   SIZE 3 / :CON SIZE3/
: PRIME  0 FLAGS SIZE 1 FILL SIZE3/
   #[ SIZE3/ I - DUP FLAGS + C@
      ?[ DUP 2* 3 + 2DUP +
         [[ DUP SIZE < ?[ 0 OVER FLAGS + C! OVER + ]]
         2DROP DROP 1+ ][ DROP ]? ]#
   3 SPACES . ." PRIMES " ;

GCD                          n1 n2 -- n3
Finds the Greatest Common Divisor of two top words.
: GCD    [[ SWAP OVER MOD DUP 0= ?] ;
```

9.9. DATES

```
CDN         dd mm yy -- cdn Top=YY, 2nd=MM, 3rd=DD.
Converts these three values to a value which is the number of days into the Century. This does not have only 28
days in February 1900, hence it is not reliable prior to 1 Jan 1901. 7 MOD of this result is the day of the week
with 0 being Sunday.
: CDN    OVER 3 < ?[ 1- SWAP 13 + SWAP ][ SWAP 1+ SWAP ]? 1461 4 */
          SWAP 306 10 */ + + ;

DMY          cdn -- dd mm yy
Converts the Century Day number to Year Month and Day. Top=Year, 3rd=day.
: DMY    DUP 122 - 4 1461 */MOD -ROT 0= 2* + OVER 1461 4 */ -
          DUP 1000 30601 */ SWAP OVER 30601 1000 */ - SWAP DUP 14 <
          ?[ 1- ROT ][ 13 - ROT 1+ ]? ;

WENDS        cdn -- n
This takes a Century Day Number and calculates the number of Saturdays and Sundays which occurred in the
century. This is used in WDAYS, the difference between the two values is accurate; the number of week-end days
may be off by a fixed amount.
: WENDS   DUP 7 /MOD 2* SWAP 0= - + ;

WDAYS       dd1 mm1 yy1 dd2 mm2 yy2
This takes two calendar dates in the CDN form and calculates the number of working days (Monday - Friday) there
has been between the two dates.
: WDAYS   CDN WENDS >R CDN 1+ WENDS R> - ;
```

9.10. ARRAYS

```
:CHAR       n --
Create a character or byte array. n bytes are allocated.
: :CHAR   :BUILD ALLOT ;: + ;

:VECT       n --
Create an n item integer array.
: :VECT   :BUILD 2* ALLOT ;: SWAP 2* + ;

:ARRAY      n1 n2 --
Create a two dimensional integer array of n1xn2 entries.
```

```
: :ARRAY  :BUILD OVER , * 2* ALLOT ;: >R I @ * + 2* R> 2+ + ;
```

:INDEX n --
Compile an array and compile n word addresses into this array.

```
: :INDEX  :BUILD #[ ' , ]# CR ;: SWAP 2* + @ ;
```

!* –
Compile address of this word into this definition. This allows recursion. It is the user responsibility to test for the exit condition; otherwise, the R-stack will overflow.

```
: !*  'LAST , ; IMM
```