

An Extensible User Interface

Almost all the Forth applications I write for research work in mathematics have one or two dozen top-level commands. I use these systems in interactive sessions: data is entered, some commands are invoked, the results are examined, then further commands are issued. New commands may be temporarily introduced during a session; as the research project evolves, new features may also be added permanently. Thus, Forth is used to provide a computing environment which is interactive and an underlying system which is flexible. The research system is extended and modified as it is used.

The present article is the outgrowth of work to prepare Forth systems for use by others. I am interested in showing my research work to other mathematicians and in integrating computer use with some of the pure mathematics courses I teach. In both cases, very few members of my intended audience know anything about computer programming, and essentially none know Forth. My earliest attempts to show my work to others involved providing supplementary written material on Forth. This would allow my applications to be used in essentially the same way I used them (see FORML 90). This approach was successful with some instructional material used in a course in which I was teaching Forth. In general, however, it assumes that people are willing to learn the basics of a computer language in order to use an application (or even to find out if the application interests them). Very few people are willing to do this. I, therefore, wanted to find a simple way to add a user interface to existing applications. There were several criteria for such an interface:

1. It should be easy to use.
2. It should be easy to add to an existing application (without requiring the application to be specially written).
3. It should be easy to extend as the underlying application is extended.
4. It should allow the user to invoke all the top-level commands.
5. It should have an integrated help system.

An interface which satisfies these criteria is described in this article. It is a *commands-completion* interface: The user sees a list of commands. He types enough letters to identify a command uniquely, and the rest of the command name is completed for him and executes the command. The user is prompted for any input needed to carry out the command. Typing the command `INFO` and then another command will provide descriptive information about the command (rather than executing it). The enclosed source code also shows an alternative: type the command name preceded by a question mark.

This interface has solved several problems. For instruc-

tional programs, it has provided students an easy way to interact with an application. They can learn to use it very quickly. It allows mathematical applications to be used when there is no time to teach programming. It allows me to modify and extend an application and interface without recompiling the code of the system. (I turn software over to the computer center at the start of a course and do not have access to it thereafter.) I can also produce optional modules which extend the interface as well as the application.

This interface may also be useful to others to allow Forth work to be shown outside the Forth community. Forth applications usually do not run “standalone.” To run an application, a Forth system is needed. This fact puts Forth at a disadvantage with respect to compiled languages. Anyone who wishes to show their Forth applications to those outside the Forth community must usually supply a Forth system with the application or force potential users to obtain one on their own. In some cases, this means they must mess with adapting source code to another version of Forth. The user interface provides an alternative: Elizabeth Rather informs me that Forth, Inc. and other vendors of commercial Forth systems allow their systems to be supplied without fee or license with turnkey applications. A Forth application with this user interface can be “turnkeyed” (i.e., headers removed, one top-level word, and the application saved as an executable). A Forth application can, therefore, be supplied in a trouble-free, load-and-run form just like applications written in compiled languages.

Example

Figure One presents an example showing the interface used for an instructional application in Group Theory. The application computes information about groups of order up to 32 (see FORML 90). In these examples, the user’s input is underlined.

Implementation

The menu names of commands are stored in a binary tree together with the execution token of the Forth word needed to carry out the command. When the user types a character, the tree is searched. If a unique entry is found, the command is completed. If no match is found, the system beeps and removes the erroneous letter. If several matches are found, the system waits for further letters.

New commands are added by `>CMD <menu_name> <Forth_word>`. The Forth word must prompt the user for information needed to carry out the command. The group table for group 8, for example, is obtained in the underlying Forth system by `“8 Table”`. A new word, `%Table`, is created which contains the help information, prompts for input of a

group number, and executes the underlying Table word. A command is added to the menu by >CMD TABLE %Table. (The commands in the tree are not part of the dictionary, so there is no problem if they have the same name as in the underlying Forth application.) See the source listing for information about the "help" and input words.

```

: %Table
  Help:
  This prints a table for the group requested
  (and makes that the current group). Elements
  are represented by letters A to Z and the symbols
  [ \ ] ^ _ and `
  Help;
  Input" for group number \Get-Num " CR Table ;
>CMD TABLE %Table

```

Figure One. Example use of interface.

```

CENTER          CENTRALIZER  CHART          CONJ-CLS
COSETS          EVALUATE     EXAMPLES       GENERATE
GROUP           HELP         INFO           ISOMORPHISM
LEFT            NORMALIZER   ORDERS        PERMGRPS
POWERS          QUIT         RESULT        RIGHT
SEARCH         STOP         SUBGROUPS     TABLE
X

```

G1>> CHART Order of Groups (1-32 or 0) Number 12
 20 21 22* 23* 24*
 There are 5 Groups of order 12
 2 abelian and 3 non-abelian

G1>> CHART Order of Groups (1-32 or 0) Number 6
 7 8*
 There are 2 Groups of order 6
 1 abelian and 1 non-abelian

G1>> TABLE for group number 8

```

  A B C D E F
A |A B C D E F
B |B C A F D E
C |C A B E F D
D |D E F A B C
E |E F D C A B
F |F D E B C A

```

G8>> INFO
 This will provide information about the next
 command you use. INFO and X do the same thing
 but X is quicker to use.

G8>> EVALUATE
 This is used to evaluate an expression in the current
 group. An expression is a collection of group elements
 and inverses which is evaluated left to right. An
 apostrophe following a letter is used to indicate the
 inverse of the letter. Thus BC'D will give the product
 of B followed by the inverse of C followed by D

```
G8>> EVALUATE (use ' for inverse) bd= F
G8>> EVALUATE (use ' for inverse) db= E
```

This system has a sub-menu of commands for permutations:

```
G8>> PERMGRPS
CREATE      ELEMENTS    HELP          INFO
INSTALL    MAIN        MULTIPLY     QUIT
X
```

```
PERM>> ?CREATE
```

This will determine the subgroup of Sn generated by a given set of permutations (given as a product of cycles). You must put in n (for Sn) and then the generators using numbers 1..n for example (1 2)(3 4 5). The program will only compute groups up to order 51. If the resulting group has order 32 or less, you can install the table as one of the groups 1-5.

```
PERM>> CREATE
```

```
Subgroup of Sn -- what is n? Number 4
Put in generators as product of cycles.
End with a blank line
```

```
Generator (1 2)(3 4)
```

```
Generator (1 2 3 4)
```

```
Generator
```

```
Group is of order 8
```

A ()	B (2 4)	C (1 2)(3 4)
D (1 2 3 4)	E (1 3)	F (1 3)(2 4)
G (1 4 3 2)	H (1 4)(2 3)	

Source Code Listing

Supplements to ANS-Forth

1. The words Comment: and Comment; can be defined in a similar way to Help: and Help; below.
2. AT (same as AT-XY) and AT? are used to set and find cursor position.
3. UPC (ch -- ch') converts a character to upper case
UPPER (addr cnt --) converts a string in place
4. DEFER and IS are used for vectored execution
5. (.") is the literal string handler put in place by ."
6. NUMBER? (addr len -- d flag)
flag is TRUE if number was properly converted
d is the double number obtained
7. The following are common:
: 3DUP 2 PICK 2 PICK 2 PICK ;
: -ROT ROT ROT ;
: NOT0= ;
: >= < NOT ;
: CELL 1 CELLS ;
: BEEP 7 EMIT ;
: OFFFALSE SWAP ! ;
: ON TRUE SWAP ! ;

Source Code

```
\ ****      Command Completion Interface      ****
\
\      John J Wavrik   Dept of Math
\      Univ of Calif - San Diego

      30 CONSTANT Max#Cmds
      16 CONSTANT CmdSize   \ make a multiple of bytes/cell
      0 VALUE      #Cmds
CmdSize CELL + CONSTANT EntrySize

comment:
  A user is presented with a list of commands and needs only
  to type enough letters to identify the command uniquely.

  New commands are introduced by >CMD <listname> <executable>
  where <listname> is the name made available to the user and
  <executable> is a Forth word to be executed. (Typically the
  executable is a Group package Forth command which has been
  supplemented by queries for input).

  The listwords are stored alphabetically in a binary tree
  to enable partial words to be easily found. Each node
  has a name (the list word) which is a string (maxsize SZ),
  and three addresses (cells): the CFA of the executable,
  and the address of left and right subtrees.
comment;

\ ****      Binary Search Tree for Strings      ****

\ Counted String Operations

: $! ( $ addr -- ) OVER C@ 1+ MOVE ; \ no test for fit
: $. ( $ -- ) COUNT TYPE SPACE ;
: $Compare ( $1 $2 -- -1 | 0 | 1 )
  \ -1 = $1 is before $2
  \ 0 = $1 equal to $2
  \ 1 = $1 is after $2
  >R COUNT R> COUNT COMPARE ;

: $< $Compare 0< ;
: $= $Compare 0= ;
: NCompare ( $1 $2 n -- -1 | 0 | 1 )
  \ compare first n characters
  \ must pad strings with blanks if n is big
  ROT 1+ ROT 1+ ROT ( addr1 addr2 n )
  TUCK COMPARE ;

Max#Cmds      CONSTANT #Nodes
CmdSize      CONSTANT SZ      \ maximum string size for names
SZ 3 CELLS + CONSTANT NodeSZ  \ size of node in bytes
0 VALUE FreeNode      \ address of free node variable
VARIABLE Len-Name      \ length of longest name

: $!! ( $ addr -- )
  OVER COUNT Len-Name @ MAX Len-Name ! DROP
  $! ;

CREATE 'Tree1 #Nodes NodeSZ * ALLOT
VARIABLE FreeNode1
CREATE 'Tree2 #Nodes NodeSZ * ALLOT
VARIABLE FreeNode2
```

```

comment:
  In this application there is a main menu (using Tree1)
  and a submenu (using Tree2) activated by a command on
  the main menu. The same idea can be used to allow multiple
  submenus.
comment;

'Tree1 VALUE Tree \ can extend to several trees

: Tree.Init Tree #Nodes NodeSZ * ERASE
  Tree FreeNode ! 1 Len-Name ! ;

\ All operations refer to the "current tree".
\ The address of the root of the current tree is
\ given by Tree. The address of the last filled
\ node is given by FreeNode

: Tree1 'Tree1 TO Tree FreeNode1 TO FreeNode ;
: Tree2 'Tree2 TO Tree FreeNode2 TO FreeNode ;
Tree2 Tree.Init
Tree1 Tree.Init

: NewNode ( -- addr ) NodeSZ FreeNode +!
  FreeNode @ DUP NodeSZ ERASE ;
  \ there is no error trap here if the tree is full

: Left ( n_addr -- l_addr ) SZ + @ ;
: Right ( n_addr -- r_addr ) SZ + 1 CELLS + @ ;
: Exec ( n_addr -- ) SZ + 2 CELLS + @ EXECUTE ;
: Left! ( x n_addr -- ) SZ + ! ;
: Right! ( x n_addr -- ) SZ + 1 CELLS + ! ;
: Exec! ( e_addr n_addr -- ) SZ + 2 CELLS + ! ;
: Name! ( $ n_addr -- ) DUP SZ BLANK $!! ;
: Leaf? ( n_addr -- flag )
  DUP Right 0= SWAP Left 0= AND ;

comment:
  Notice that we assume (and use) the fact that the name
  of a node is stored at the address of the node -- while
  pointers are stored at offsets from this name address.

  Notice also that storing a name (by Name!) pads the
  name with blanks -- to allow use of NCompare
comment;

DEFER (>Tree) \ this allows recursive definition for
  \ storing a new name in the tree
: Go-Left ( $ n_addr -- ) DUP Left
  IF Left (>Tree) ELSE
  SWAP NewNode TUCK Name!
  SWAP Left! THEN ;
: Go-Right ( $ n_addr -- ) DUP Right
  IF Right (>Tree) ELSE
  SWAP NewNode TUCK Name!
  SWAP Right! THEN ;
: (>Tree)-AUX ( $ n_addr -- )
  DUP C@ 0= IF Name! ELSE
  2DUP $Compare
  DUP -1 = IF DROP Go-Left ELSE
  1 = IF Go-Right ELSE
  ( 0 = IF) 2DROP THEN THEN THEN ;
' (>Tree)-AUX IS (>Tree)

```

```

\ Put a new name in the tree -- eventually the execution
\ address will be stored also. Note that this does not
\ store duplicate names.
: >Tree ( $ -- ) Tree (>Tree) ;
\ Given a string $, count n, and node address n_addr
\ Find a node in the subtree with root at n_addr so
\ that the name matches the string up to n characters

: (NFind) ( $ n n_addr -- n'_addr t | f ) DUP 0=
  IF DROP 2DROP FALSE ELSE
    3DUP SWAP NCompare
    DUP -1 = IF DROP Left RECURSE ELSE
      1 = IF Right RECURSE ELSE
        >R 2DROP R> TRUE THEN THEN THEN ;
: NFind? ( $ n n_addr -- t | f )
  (NFind) DUP IF SWAP DROP THEN ;
\ See if a string matches the first n characters of
\ some node in the tree. Indicate if multiple match
: NFind ( $ n -- n_addr -1 | n_addr 1 | f )
  \ -1 = more than one match
  2DUP Tree (NFind) ( $ n addr t | $ n f )
  IF >R 2DUP R@ Left NFind? -ROT
    R@ Right NFind? OR
  R> SWAP IF -1 ELSE 1 THEN
  ELSE 2DROP 0 THEN ;

: Node.L ( node -- ) ?DUP IF 2 SPACES COUNT DROP
  Len-Name @ TYPE
  THEN ;

: CR_4 ( cnt -- cnt' ) ?DUP 0= IF CR 4 THEN 1- ;

: (Print-Nodes) ( cnt tree -- cnt' ) ?DUP
  IF DUP Leaf? NOT
    IF TUCK Left RECURSE
      OVER Node.L CR_4
      SWAP Right RECURSE
    ELSE Node.L CR_4 THEN
  THEN ;

: Print-Nodes CR 3 Tree (Print-Nodes) DROP ;

\ **** Keyboard Input Routines ****

VARIABLE Tfound VARIABLE TAddr
8 CONSTANT BS 7 CONSTANT BELL 27 CONSTANT ESC 127 CONSTANT DEL

\ ClrKey
\ If the user types in more characters than needed
\ to complete a command, this clears the extra characters
\ from the keyboard buffer.

: ClrKey BEGIN KEY? WHILE KEY DROP REPEAT 30 MS ;

\ Del-In Do-ESC
\ The following are actions to be taken by BS or DEL
\ and ESC. n is the number of characters so far in the
\ input word. c is an arbitrary character (it is dropped
\ but included for compatibility with other action words)

: Del-In ( n c -- 0 | n-1 )
  DROP DUP IF 1- BS EMIT SPACE BS
  ELSE BELL THEN EMIT ;

```

```

: Do-ESC ( n c -- )
  DROP TFound ON TAddr OFF
  DUP 0 ?DO 0 Del-In LOOP
  ." *** cancelled *** " CR ;

: Check-Tree ( a n char -- a n+1 ) \ sets tfound
  3DUP EMIT + C! 1+ ( a n+1 )
  OVER 1- ( $ ) OVER NFind
  DUP 1 = ( unique ) IF DROP TFound ON TAddr ! ELSE
  0= ( none ) IF BELL EMIT BS Del-In ELSE
  ( several ) TFound OFF DROP THEN THEN ;

\ Notice that characters from keyboard are uppercased
VARIABLE Help? VARIABLE FirstChar

: TExpect SZ PAD 1+ \ get characters until found in tree
  0 ( len adr 0 ) TFound OFF TAddr OFF FirstChar ON
  BEGIN 2 PICK OVER - ( len adr #so-far #left )
  0<> TFound @ 0= AND

  WHILE KEY UPC ( len addr #so-far char )
    DUP [CHAR] ? = FirstChar @ AND
    IF EMIT Help? ON ELSE
    DUP BS = IF Del-In ELSE
    DUP DEL = IF Del-In ELSE
    DUP ESC = IF Do-ESC ELSE
    DUP BL > IF Check-Tree ELSE
    DROP THEN THEN THEN THEN THEN
    FirstChar OFF
  REPEAT DUP 0 ?DO BS EMIT LOOP 2DROP DROP
  ClrKey
  TFound @ IF TAddr @ $. 2 SPACES THEN
  TFound @ 0= ABORT" character count exceeded " ;

: CExpect ( -- )
  TExpect TAddr @
  ?DUP IF Exec THEN ;

\ **** Command Completion Module ****

\ Notice that command names are uppercased

: >CMD ( -- ;;; follow by <name><action> )
  #Cmds Max#Cmds >=
  IF ." Command list is full " CR BEEP
  ELSE BL WORD DUP COUNT UPPER
    DUP >Tree
    DUP C@ NFind 1 =
      IF ' SWAP Exec! ELSE
      TRUE ABORT" Error in insertion " THEN
  THEN ;

\ ***** a Help System for Command Words *****

: Make, ( delimiter -- )
  \ Defining word for words that compile input
  \ string up to delimiter.
  CREATE ,
  DOES> @ PARSE HERE >R DUP C, DUP ALLOT
  R> 1+ SWAP MOVE 0 C, ALIGN ;

0 Make, ,0 \ compile entire line as counted string
CHAR " Make, ," \ compile up to a quote

```

```

CHAR \ Make, ,\ \ compile up to a backslash

comment:
  The words Help: and Help; are used to bracket text
  which describes what a command does and/or how it is
  used. This text is put at the start of a definition.
  If the user presses X or type INFO before a command,
  this information is displayed instead of having the
  command action carried out. Help: and Help; should
  be at the start of new lines with the descriptive
  text on lines between (just as "comment:" and "comment;"
  are used to bracket the current paragraph).
comment;

: HelpX 0 Help? ! ;

: Help:      ( -<text> Help;>- )      \ the word Help; must start
      \ a new line
      POSTPONE Help? POSTPONE @ POSTPONE IF
      BEGIN  >IN @ BL WORD DUP COUNT UPPER
      COUNT S" HELP;" COMPARE 0=
      IF DROP TRUE
      ELSE >IN ! POSTPONE (".") POSTPONE CR
      REFILL 0=
      THEN
      UNTIL
      POSTPONE HelpX
      POSTPONE CR POSTPONE EXIT POSTPONE THEN ; IMMEDIATE

: %INFO CR
  ."      This will provide information about the next" CR
  ."      command you use. INFO and X do the same thing" CR
  ."      but X is quicker to use." CR
  -1 help? ! ;

\ ****      Main Loop      ****

: %END
  CR
  ."      This will end the command interface (but not the" cr
  ."      groups program). You can resume use of the commands" cr
  ."      interface by typing 'commands'." cr cr
  ."      *** Exit the program by typing `bye` *** " cr
  DROP
  TRUE ABORT" ++++++ " ;

: %Help
  Help:
  This prints a list of all current commands
  Help;
  CR Print-Nodes CR ;

\ Commands
\      This is the top level word used to start
\      the interface

: Commands FALSE %Help
  BEGIN
    CR ." >> "
    ['] CEXPECT CATCH DROP DUP
  UNTIL DROP ;

```

```

Tree1 Tree.Init
>CMD INFO %INFO          >CMD X      %INFO
>CMD STOP %END           >CMD QUIT %END
>CMD HELP %Help

\ ****      Commands for prompted input      ****

\ Get-TIB
\ This is a word which gets (and edits) keyboard input until terminated by
\ pressing ENTER.  The input must be placed at the start of the terminal
\ input buffer. The buffer pointer is reset.  The input should be displayed
\ right after the prompt. When Get-TIB is finished the cursor should be
\ right at the end of input. ANS standards do not specify the display and
\ editing actions for ACCEPT -- so some systems may require a custom version.

: Get-TIB ( -- )
  AT? QUERY AT          \ Put cursor at end of prompt
  >IN @ 0 WORD          \ Put cursor at end of input
  COUNT TYPE >IN ! ;

\
\ ****      Samples for prompted input      ****
comment:
  The following prompted input words are included as samples.
  An input word should be designed for each type of data. It
  should provide a prompt; get an input line (using get-TIB);
  process the line and perhaps check for validity; and leave
  on the stack whatever the action word expects to find.
  Invalid input can either throw an exception or discard the
  invalid input to allow the user to try again.
comment;

2VARIABLE Save-Pos
: Get-Num ( -- n )
  AT? Save-Pos 2!
  BEGIN Save-Pos 2@ AT          \ reposition to start
  Get-TIB BL WORD
  COUNT ?DUP 0= THROW          \ empty input aborts the command
  NUMBER? IF DROP TRUE
  ELSE 2DROP BEEP FALSE \ invalid input starts over
  THEN
  UNTIL ;

\ Fancy input routine

: Pos ( char -- pos ) \ pos = 0 if not found
  >IN @ SWAP PARSE 2DROP
  >IN @ #TIB @ > ( past end of buffer )
  IF 0 ELSE >IN @ THEN
  SWAP >IN ! ;

: Input"
  BEGIN
    [ CHAR] \ Pos
    IF POSTPONE (." ) ,\
      BL WORD DUP COUNT UPPER
      FIND 0= ABORT" word not found"
      COMPILE, FALSE
    ELSE
      [ CHAR] " Pos
      POSTPONE (." ) ,"
    THEN
  UNTIL ; IMMEDIATE

```