A **Fifth** Tutorial

The Software Construction Company, Inc

September 19, 1989

1 The Fifth Environment

Fifth is a browser–based system in the tradition of workstation–class Smalltalk and Lisp environments. This differentiates **Fifth** from more familiar integrated systems such as the Borland and JPI environments, where the primary operating mode is the text editor. **Fifth** is also a highly interactive language, providing a command line interface that allows you to quickly test and debug your code.

1.1 The Fifth Browser

The browser adds an extra dimension to program editing. Rather than a flat file of text, the browser manipulates code at the functional level, allowing functions to be visually moved, grouped, and scoped. It supports programming in the large, providing the higher–level development facilities such as turnkey generation and an automatic "make." Here are the commands available from the browser:

<*ESC*> – Exits the Browser.

<←> – Move left one child.

<→> – Move right one child.

<↑> – Move up one level.

<↓> – Move down one level.

<*Ctrl* ←> – Move the currently selected child left.

<*Ctrl* →> – Move the currently selected child right.

<*Ctrl-PgUp*> – Move the currently selected child up one level.

<*Ctrl-PgDn*> – Move the currently selected child down one level.

<*A*> – Make a TURNKEY application.

<*C*> – Compile the local module.

<*E*> – Edit the local module.

<*D*> – Delete the selected module.

<*G*> – (Go) Execute the local module.

<*I*> – Insert a new module in front of the selected module. Prompts for the module name.

<*L*> – Load a file previously saved using <S>.

<*Q*> – Quit to MS-DOS.

<*R*> – Renames the local module. Prompts for the new name.

<*S*> – Saves the local module and all of its children to disk. Prompts for a file name to save the module in.

1.2 The Fifth Text Editor

The editor plays a somewhat subordinate role in the **Fifth** system because the browser is used to edit at the module level. **Fifth** encourages very modular programming; thus, the text editor is very simple, and is designed for editing small amounts of text. As a rule of thumb, if the editor becomes cumbersome, your modules are probably too large, and should be factored into several smaller modules. The following is a list of commands available in the editor:

<*Home*> – Move to the start of the current line.

<*End*> – Move to the end of the current line.

<*PgUp*> – Move up 23 lines.

<*PgDn*> – Move down 23 lines.

<*Ins*> – Toggle between insert and overstrike modes.

<*Del*> – Delete the character under the cursor.

<*Ctrl-PgUp*> – Drag the current line up one line.

<*Ctrl-PgDn*> – Drag the current line down one line.

<*ESC*> – Exits the Editor, returns to the previous environment.

<*TAB*> – Skip forwards to next tab stop.

<*Shift-TAB*> – Skip backwards to previous tab stop.

<*Ctrl-C*> – Abort the edit session, changes are lost.

<*Ctrl-E*> – Scroll up 1 line.

<*Ctrl-G*> – Delete the character under the cursor.

<*Ctrl-N*> – Split the line at the cursor position.

<*Ctrl-W*> – Toggle word wrap mode.

<*Ctrl-X*> – Scroll down 1 line.

<*Ctrl-Y*> – Delete the current line.

<*F1*> – Call up help on the editor.

<*F2*> – Call up help on the module under the cursor.

<*F3*> – Insert blank line into text.

<*F7*> – Delete the current line.

<*F8*> – Undelete the line last deleted.

<*F9*> – Split the current line at the cursor position.

<*F10*> – Join the current line with the next line.

<*Backspace*> – Delete the character to the left of the cursor.

<*Return*> – Move to the start of the next line.

1.3 The Fifth Command Line Interface

The browser and editor are only half of the story, however. Much of **Fifth**'s power comes from its interactive mode. In the interactive mode, you can enter just about any legal **Fifth** expression and have it evaluated instantly, making testing and debugging painless.

1.4 Debugging

Debugging in **Fifth** is both simple and efficient. **Fifth** provides a primitive module, `trace`, that when executed prints the name of the module you are about to execute and the stacks, and pauses until a key is pressed. For more dirty debugging jobs, there is a command available in the Browser that will toggle single–step mode on a given module. A module in single–step mode will do a `trace` between every module it calls as it executes. This mode is very useful for finding stack errors in your code. As an added feature, any time **Fifth** is waiting for a key in `trace`, you can press the <Return> key to shell to the command line. Here you have the full power of **Fifth** to help you explore your program.

1.5 The Help System

As an aid to those of us without eidetic memory, there is a context-sensitive help system available in all of **Fifth**'s operating modes. Since the help system covers **Fifth** in nearly as much detail as the manual, it is usually all the documentation that an experienced **Fifth** programmer needs. For example, in the editor you can place the cursor on the name of any module and press the <F2> key for help. If the module is not a primitive, then you are taken to its source code instead. Either way, you get information on what the module does. This system drastically reduces the amount of time wasted in flipping through the manuals for documentation on some obscure command.

1.6 The Fifth Compiler

The **Fifth** language is a 32–bit descendant of Forth–83. Its extensions include automatic incremental compilation, built in 80x87 coprocessor support, DOS support, and a handle-based memory management system. **Fifth** compiles to optimized 8086 code for fast execution, and the professional version of **Fifth** is also capable of producing conventional DOS .EXE turnkeys.

1.7 The Fifth Data Stack

The data *stack* is Fifth's central data structure. It is used for evaluating expressions, passing parameters, and temporary data storage. It is impossible to program in Fifth without a solid understanding of its data stack.

The classic model for a stack is a rack of cafeteria trays in a spring–loaded holder. As trays are placed on top of the rack, the other trays are pushed down. Since trays can only be removed from the top of the holder, the first trays placed on the stack will be the last trays removed (often referred to as LIFO, meaning Last–in, First–Out).

This stack–based system is not as unusual as it may seem. Most programming languages, including *C* and *Pascal* use a stack to evaluate their expressions. This translation from an infix expression system to a postfix, stack–oriented system is done by the *C* or *Pascal* compiler. **Fifth**, however, lets the programmer work directly with the stack. This allows the programmer to write operators that take more than two operands (impossible in *C* or *Pascal*), and functions that return more than one result. This leads to a style of programming called *functional programming*, where a function is more for the transformations it makes on the stack than for any side effects.

Despite postfix's simplicity and power, the fact remains that most of the world uses the algebraic form for expressions. It is occasionally necessary, therefore, to convert an infix expression to its equivalent postfix form. Fortunately, this is a straightforward mechanical procedure which is quite easily mastered, as many Hewlett–Packard calculator users can attest. The procedure for converting an infix expression to postfix goes as follows:

1. Scan the infix expression from right to left, moving operators behind their operands (an expression enclosed in parentheses is really a single operand).

2. Within each pair of parentheses, do step (1) and remove the parentheses.

3. Be careful of the precedence of such operators as multiplication (*), division (/), and exponentiation (^).

The best way to learn this procedure is to just jump in and actually try it. Here is a simple example.

```
(5 + 95) / 10
```

The topmost operator is the division (/) operator. It's two arguments are '`(5 + 95)`' and '`10`'. So the first transformation is to move the '`10`' in front of the '/' operator, like this:

```
(5 + 95) 10 /
```

Now it is necessary to process the parenthesized expression '`(5 + 95)`'. It is transformed in the same way as the outer expression, but this time there are only simple (non-parenthesized) operands. The final result is the postfix expression

```
5 95 + 10 /
```

Now you are ready for a more complicated example. Here is the transformation of the expression '`(15 - 10) * (3 + 1) / 4`'.

```
(15 - 10) * (3 + 1) / 4
(15 - 10) (3 + 1) *   4 /
 15 10 -   3 1 +  *   4 /
```

Notice that the order of the operands is the same in both the infix and postfix forms of the expression. The only difference between the two types of expressions is that the operators have been moved around and all parentheses have been removed. This makes it easy (with a little practice) to convert an infix expression to its postfix equivalent in a single pass. Here are some more infix expressions to help you gain experience in converting to postfix:

1. `((1 + 5) - (3 - 2)) * 5`
2. `(1 + 2 + 3 + 4 + 5) / 15`
3. `15 / (1 + 2 + 3 + 4 + 5)`
4. `((((5 * 4) / 3) * 2) / 1)`
5. `1 + 2 * 3 - 4 / 5 * ((6 - 7))`
6. `((1 - 2) * ((3 - 4) + (5 + 6)) / (7 / 9))`
7. `(1 + (2 - (3 + (4 - (5 + 6)))))`

Try to work the problems before looking at the answers below.

1. `1 5 +  3 2 -  -  5 *`
2. `1 2 + 3 + 4 + 5 +  15 /`
3. `15  1 2 + 3 + 4 + 5 +  /`
4. `5 4 * 3 / 2 * 1 /`

```
5. 1  2 3 * +  4 5 / -  6 7 - *

6. 1 2 -  3 4 -  5 6 +  +  *  7 9 /  /

7. 1  2  3  4  5 6 +  -  +  -  +
```

2 The Fifth Tree

Fifth source code is maintained by the browser in a highly structured tree–like hierarchy. In this scheme, **Fifth** is at the top of the tree (the "root") and all of the primitives and user–written modules are children of **Fifth**.

2.1 Scoping

The **Fifth** compiler uses this tree structure to provide scoping to a program. When **Fifth** searches for a name, it starts at the children of the current module in the tree and searches backwards through the names on the current level. When it reaches the beginning of the list, it goes up to the previous level, and begins searching back from there. Figure 1 shows how such a search would proceed.

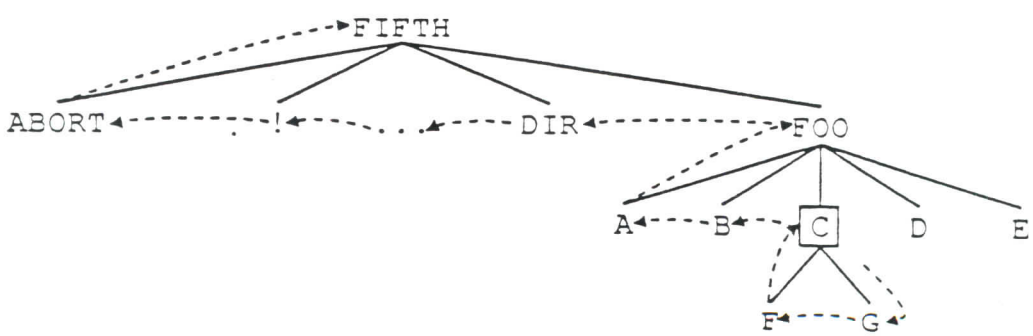

Figure 1: Example search

2.2 Packages

A pure tree–structured approach has several problems, as evidenced by languages like Pascal, since it makes no provision for libraries of frequently used routines. To break the standard scoping rules when they prove too restrictive, **Fifth** provides *packages*. This is a collection of modules hidden in a special module that allows them to be found by **Fifth** during a name look-up. **Fifth** uses packages to hide libraries of related functions like graphics or floating point operations.

3 A Session With Fifth

So far, we have only *talked* about **Fifth**. Now it is time to actually get in and *do* something with it. Over the next few pages we will develop a small program, `DUMP`, that will dump a binary file to the screen in a comprehensible format.

3.1 Introduction to `DUMP`

However, first we will jump ahead a bit and give present a hierarchy chart (figure 2) for the final program.

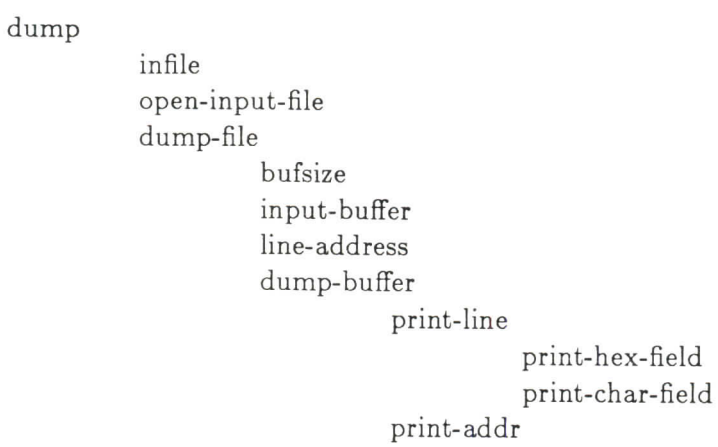

Figure 2: Functional hierarchy of `DUMP`

This functional hierarchy corresponds nicely to its **Fifth** tree representation, as shown in figure 3:

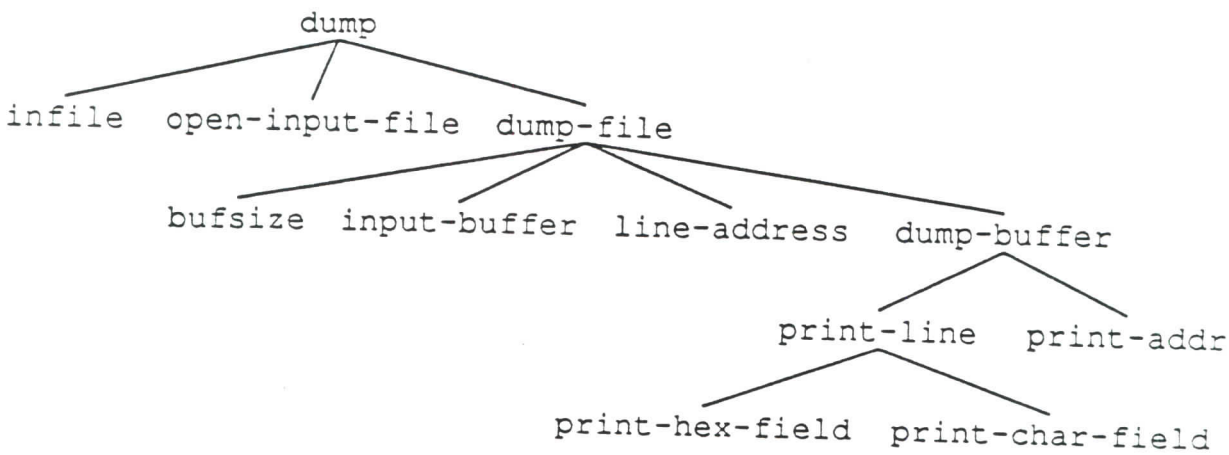

Figure 3: Fifth tree structure for DUMP

3.2 Developing the Code

We will be developing DUMP in a top-down manner, as befitting a structured language like **Fifth**. At the highest level, all DUMP needs to do is open the file, print a dump somehow, and close the file. Since both opening and dumping the file are likely to be several lines of code, we will push them off into their own modules. Closing the file is very simple, so we will just code that in line.

```
\ This is a small program that reads a filename from the
\ command line and dumps that file to stdout in a nice,
\ readable format.
\ SYNTAX: DUMP filename
: dump  ( -- )
  open-input-file
  dump-file
  infile @ close
;

\ holds the handle for our input file
variable infile
```

But before you go and type this code in, we should probably cover some of the errors that beginning **Fifth** programmers commonly make. First of all, most of the white space in this program *is* significant. **Fifth** uses white space to delimit words in the code. This means that both `:` and `dump` are words **Fifth** can recognize, but `:dump` is not. When **Fifth** finds a word that it cannot recognize, it pops up an error box containing the offending word and a short descriptive error message.

Since we are developing DUMP top-down, when you leave the editor you will get an error message informing you that **Fifth** could not find `open-input-file`. Don't worry, just press <Return> to get back to the Browser. We will be adding this and other modules later in this section, as we work through the details of DUMP.

Another common error is forgetting the semicolon at the end of `:` defined modules. Since many **Fifth** primitives are mere punctuation marks, it is easy for your eye to simply skip over them as you read.

Notice that, although the `infile` variable is declared in the same listing as the code for `dump`, it must be placed in a module of its own. After typing in `dump` and `infile`, the screen should look something like

---

```
[DUMP] F1 - Help, Trace is OFF, Memory: 411792

INFILE
=================================================================
\ This is a small program that reads a filename from the
\ command line and dumps that file to stdout in a nice,
\ readable format.
\ SYNTAX: DUMP filename
: dump  ( -- )
  open-input-file
  dump-file
  infile @ close
;
=================================================================
\ holds the handle for our input file
variable infile
```

Having written this, our next step is to flesh out the details of `open-input-file`. This module takes advantage of a few useful features of **Fifth**. When a **Fifth** turnkey program starts running, **Fifth** loads the `pad` buffer with the text of the command trailer. By happy coincidence, **Fifth** also initializes the input pointer to the start of the `pad` buffer. This

means that we can use `word` to parse out the filename. But **Fifth** also uses the `pad` buffer for parsing keyboard input in interactive mode. This means that you can run DUMP by typing

```
DUMP filename
```

at either the DOS prompt or in **Fifth** interactive mode. This makes it very easy to test DUMP during development.

```
\ Open the file, place the handle in INFILE for use by
\ the rest of the program.
: open-input-file       ( -- )
  32 word dup c@ if         \ parse out the filename
    1+                      \ step over the count byte
    0 open 0=               \ open the file
    abort" File not found" \ abort it error
    infile !                \ save away our handle
  else          \ the user didn't give us a file name
    ."   Usage: DUMP filename" cr cr
    ."   Dumps a binary file in a readable format."
    cr
    abort
  endif
;
```

`Dump-file` also has a few quirks, since it must correctly handle buffering and any conditions caused by the `read` module. Notice how `input-buffer` is created. This is the usual way of creating a **Fifth** array. Since we will use the buffer size in several places in the code, we should put it into a named constant. One thing to be aware of is that **Fifth**, unlike Forth, does not compile a call to a named constant whenever it is encountered. One ramification of this is that modules using the constant must be forcibly recompiled whenever the module changes (simply recompiling its parent usually suffices).

```
\ does the actual dumping of the file.
: dump-file     ( -- )
  0 line-address !  hex
  begin
    input-buffer BUFSIZE infile @ read
    0= if ." DOS error #" . abort endif
  dup while              \ stop when we read 0 bytes
    input-buffer dump-buffer
  repeat                 \ keep processing the file
;

1024 constant BUFSIZE    \ size of disk buffer

create input-buffer BUFSIZE allot \ allocate the buffer

variable line-address    \ contains the line counter
```

`Dump-buffer`'s responsibility is to run through the input buffer, dumping out 16 bytes to a line. It must also maintain the file index independently of the current buffer and line.

```
\ dumps a buffer to the screen.
: dump-buffer   ( bytes addr -- )
    begin
      over 0 >     \ stop when the buffer is exhausted
    while
      line-address @ print-addr
      ." - "
      stack ab|baba
      16 min print-line
      16 - swap 16 +        \ bump address & count
      16 line-address +!    \ bump file offset
    repeat       \ keep processing the buffer
    drop drop
;
```

Again, any details that would tend to complicate matters have been pushed down into a lower module. If you noticed, `dump-buffer` buries the details of actually printing the file index and the output line in the modules

`print-addr` and `print-line`. The same is true in `print-line`, which buries the details of printing the hex dump and the character dump in the modules `print-hex-field` and `print-char-field`. This makes it easy to change their behavior without affecting the rest of the program. In order to be `print-line` as generic as possible, it takes an address to start reading from, and the number of bytes to dump. This makes it easy to incorporate `print-line` into, say, a memory dump routine for use in debugging **Fifth** programs.

```
\ given an address and the number of bytes to dump
\ print out a nice dump line.  It assumes that the offset
\ of the line w/in the buffer has already been printed.
: print-line    ( addr count -- )
  ?dup 0= if            \ zero count, just exit cleanly
    drop exit
  endif
  dup 16 > abort" DUMP COUNT OUT OF RANGE"
  stack ab|abab         \ keep a copy of of our parms
  print-hex-field
  print-char-field
;
```

`Print-hex-field` and `print-char-field` are quite similar to each other. Indeed, their primary difference is in the way they output each memory byte. Their sole complication is that they will pad out to the full 16–byte field if they are asked to dump less than 16 bytes of data.

Because these modules are so similar, it would seem to be a good practice to factor out their common functionality into one module. However, these are very small modules, and their differences are scattered about in their code. It turns out that after factoring out common code fragments, the code size does not change appreciably, while the resulting code is significantly less readable. `Print-hex-field` and `print-char-field`, while large by Forth standards, are still small enough to be read and comprehended at a glance.

```
\ print 16 bytes as hex
: print-hex-field       ( addr count -- )
  dup >r                 \ keep a copy of our count
  0 do
    count                \ get our byte, increment ptr
    <# # # #> type space \ print its hex value
  loop drop
  16 r> - dup 0 > if     \ pad our field to full width
    0 do
      ."     "
    loop
  else
    drop                 \ just clean up the stack
  endif
;

\ Print 16 bytes as characters.
\ Convert nonprintables to '.'
: print-char-field      ( addr count -- )
  dup >r                 \ keep a copy of our count
  ."    |"               \ print our header
  0 do
    count                \ get our char, increment ptr
    dup 32 < over 127 > or if   \ fix nonprinting chars
      drop ' .
    endif
    emit                 \ print our char out
  loop drop
  16 r> - dup 0 > if     \ fill up the rest of the line?
    0 do
      space
    loop
  else
    drop                 \ just clean up the stack
  endif
  ' | emit cr
;
```

The last routine in our little dump program is a module to print out the

file index. Although `.` is normally just fine for printing out numbers, in this case we need to print out the number in a full eight-character field with leading zero's. This requires use of **Fifth**'s numeric formatting features. In **Fifth**, as in Forth, these are `<#`, which marks the start of a numeric formatting sequence, `#`, which formats a single digit from the number, `hold`, which inserts a literal character into the formatted output, and `#>`, which marks the end of the formatting sequence. In `print-addr`, we will use only `<#`, `#`, and `#>`. Within the `<#` ... `#>` pair, we simply loop eight times, formatting a new digit each time. The result is a pointer to the formatted string, and the length of the string, exactly what is required by `type`. The string is printed out, and we are done.

```
\ formats and prints a 32 bit number
: print-addr    ( n -- )
   <# 8 0 do # loop #> type
;
```

3.3 Finale

And there you have it: a real **Fifth** application ready to go. Programming in **Fifth** goes much faster than would seem from this example. It took much longer to write about DUMP than to write the actual **Fifth** code.

If you have the professional version of **Fifth**, you can make it into a ready-to-run .EXE by making `dump` the context module, and pressing <A> to invoke the turnkey application generator. This will produce a file named DUMP.EXE on the disk. Good luck, and happy Fifth'ing!.

4 Glossary

The following is a brief description of the **Fifth** primitives used in this tutorial.

! (n addr –) Pronounced "Store," this is the counterpart to `@`. It stores the second item on the stack into the address specified by the number on top of the stack.

(n – n') Pronounced "Sharp," this module is the workhorse of the Fifth numeric formatting system. It takes a number on top of the stack and strips out the low order digit, then converts that digit to ASCII and places it into the formatted output string.

#> (n – count addr) Tosses the number on top of the stack (presumably the one being formatted), and returns the length and address of the formatted string.

+ (n1 n2 – n1+n2) Takes two 32–bit numbers from the top of the stack, adds them, and pushes the result back onto the stack.

+! (n addr –) Roughly equivalent to *C*'s "+=" and "-=" operators. It increments the value at *addr* by *n*.

- (n1 n2 – n1-n2) Takes two 32–bit numbers from the top of the stack, subtracts them, pushes the result back onto the stack.

." ... *"* (–) Pronounced "dot–quote." At compile–time, this module parses out text following it up to the terminating quote. A space is required immediately following the initial `." `, but it is not part of the string. At run–time, the parsed string is printed to the standard output device.

0= (n – flag) Tests the number on the top of the stack against zero. If the number equals zero, a true flag (-1) is left, otherwise a false flag (0) is left on the stack.

1+ (n – n+1) Increments the number on the stack by one. This module is exactly equivalent to `1 +`, but is faster to type, and compiles to slightly more efficient code on some hardware platforms.

: (–) This module is used to create executable modules. At compile–time it parses out the name of the module, and turns on the compiler.

; (–) This module turns off the compiler. `:` and `;` are used together to build an executable module.

< (n1 n2 – flag) Compares the top two elements on the stack and returns true (-1) if the second element is arithmetically less than the top element, otherwise it returns false (0).

<# (n –) Sets up a numeric formatting buffer for use by `#`, `hold`, and `#>`.

> (n1 n2 – flag) Compares the top two elements on the stack and return true (-1) if the second element is arithmetically less than the topo element, otherwise it returns false (0).

>r (n –) Pronounced "to–r." Takes the top element on the data stack and pushes it onto the return stack. This is a dangerous move since **Fifth** keeps subroutine return addresses here. However, with disciplined use, `>r` and `r>` can greatly simplify code without having to resort to variables.

?dup (n – n n) or (n – n) Pronounced "query–dup," this module has a very specialized purpose in **Fifth**. A fairly common action is to compare a number against zero, and then perform some action if the number is not zero. `?dup` helps to simplify the code in such a situation. If the number on top of the stack is not zero, `?dup` acts just like `dup`. However, if the number *is* zero, then `?dup` does nothing.

@ (addr – n) Pronounced "fetch." `@` pops the address off the top of the stack, and fetches the thirty-two–bit number at that address to the top of the stack.

(–) Treats the text up to the end of the current line as a comment.

` ( – n) At compile–time, `'` reads the next character in the input stream stream and at run–time pushes that character's ASCII value to the stack.

abort (–) Stops **Fifth**'s execution. It resets all stacks and the video mode. The compiler is turned off and the radix is set to decimal. It redirects standard input and standard output to the CON: device.

All other files are closed. All partially compiled modules are marked as uncompiled. All modules allocated with `new` are returned to free memory. The heap is compacted. The math coprocessor is reset. In a turnkey, the program is unloaded and control returned to DOS via the standard exit mechanism.

abort (flag –) Pronounced "abort–quote." This module tests the flag at the top of the stack. If the flag is non–zero, the following message is printed and `abort` is called. Otherwise the message is skipped and execution proceeds normally.

allot (n –) Takes a number off the top of the stack, and allocates that many bytes to the module currently being compiled.

begin (–) Marks the start of an indefinite loop. Either a `while` or an `until` ends the loop.

c@ (addr – 8b) Pronounced "see-fetch." `c@` pops the address off the top of the stack, and fetches the eight–bit number at that address to the top of the stack. The number is *not* sign–extended.

close (n – -1) or (n – n' 0) This module closes a MS-DOS file handle. If an error occurred, it returns the error code and a false flag, otherwise it just returns a true flag.

constant At compile–time: (n –); At run–time: (– n). At compile time, `constant` pops a number off the top of the stack, and creates a named constant with that value. At run time, that named constant pushes its value to the data stack.

count (addr – addr+1 8b) Roughly equivalent to "`*ptr++`" in *C*. It takes the address at top of the stack, fetches the byte at that address, and increments the address. This module is useful for retrieving the count byte from a string (hence the name), and for looping through an array of characters.

cr (–) Sends a CR/LF to the standard output device.

create (–) Used to create a new module in **Fifth**. It parses the following word from the input stream to name the module. It also makes the newly created module the local module. Executing the new module

will leave the address of its parameter area on the stack. `create` is usually used along with `allot` to create named arrays.

do (limit start –) Provides a fast iterative loop. `Do` expects the loop start value and ending value on the data stack. The loop will iterate from start to limit-1.

drop (n –) Tosses the top value from the stack.

dup (n – n n) Pushes a copy of the top stack value.

else (–) Separates the two parts of an `if` ... `else` ... `endif` structure.

emit (8b –) Prints the character whose ASCII value is on the top of the stack.

endif (–) Terminates an `if` ... `else` ... `endif` structure.

exit (–) Causes an immediate return from from the current module. This is used by `;` to generate a subroutine return.

hex (–) Changes the current numeric conversion radix to base–16.

hold (8b –) Inserts the ASCII value on top of the stack into the number output conversion string. This is useful for adding things like decimal points, dollar signs, etc to the numeric conversion string.

if (flag –) Tests the flag on top of the stack. If the value is non–zero, it executes the code immediately following the `if`. If the value is zero, it executes the code following the optional `else` or the required `endif`.

loop (–) Terminates a `do` ... `loop` loop. It increments the loop index and compares it to the terminating value. If the loop is not finished, it branches up to the enclosing `do` statement. If the loop is finished, it removes the index and limit from the return stack and execution proceeds normally.

min (n1 n2 – n) Returns the smaller of the top two numbers on the stack.

open (addr code – n -1) or (addr code – n 0). Treats the second number on the stack as a pointer to a null–ended string (no leading count byte). It treats the top number as a set of access flags (read–only, read/write, etc) and attempts to open an existing file. If it succeeds, it return the handle and a true flag. If the open fails, it returns the error code and a false flag.

over (n1 n2 – n1 n2 n1) Makes a copy of the second stack element to the top of the stack.

r> (– n) Pronounced "r-from." Pops the top element from the return stack and pushes it to the data stack.

read (addr count handle – n -1) or (addr count handle – code 0) Attempts to read *count* bytes from *handle* into *buffer*. If it succeeded, it returns the number of bytes actually read and a true flag. If there was an error, it returns the error code and a false flag.

repeat (–) This module terminates a `while` loop.

space (–) Outputs a space to the standard output device.

stack `stack` is a generalized stack manipulation module. It treats the text following it as a before ... after picture of the stack, and generates the code to make the transformation. For example, `stack abc|cba` reverses the top three elements on the stack.

swap (n1 n2 – n2 n1) Reverses the top two stack elements.

trace (–) Acts as a breakpoint in a code module.

type (addr count –) Prints the *count* bytes at *addr* to the standard output device.

variable At compile–time: (–), at run–time: (– addr) Used to create a named variable. At run–time, execution of that name pushes the address of that variable's data area.

while (flag –) Used to detect the termination of `while` loops. If the flag on the top of the stack is true, control remains inside the loop. If the flag is false, control transfers to the point following the `repeat`.

word (char – addr) `Word` parses the input stream on whitespace and *char* to find the next token. It leaves the address of the length byte. The delimiter is not part of the parsed token.