

Program 5: FORTH *fib* Debug and Dump

CS 636/338—Due June 2005

The instructions in this assignment are somewhat general because they are trying to cover three different implementations of FORTH. As you work through them, if one technique does not work as described, try the next technique. If nothing works, experiment a little; ask for help.

1. Debug mode.

- (a) Turn on FORTH and load the file “lexdata.f” that you used to test the FORTH lexer. This is a program that outputs the Fibonacci sequence. If you have trouble loading, use your mouse to pick up the program from your editor window and paste it into the FORTH window.

- (b) Execute:

```
PRINTING ON ( if you are using a non-networked computer with a printer.)
xxxx      ( Executing an undefined word clears the parameter stack. )
debug fib
7 fib .
```

Keep hitting the ENTER key to single-step through your program, they type PRINTING OFF.

If PRINTING ON does not work for you, use PRINT SCREEN or use your mouse to pick up the output and paste it into a printable file. On the listing, write brief comments explaining what you see. Locate the representations of TRUE and FALSE.

- (c) Leave debug mode (hopefully) by typing q.

2. Decimal and hexadecimal I/O modes.

Enter the following commands, capture the output, and write comments on the listing that explain the results.

```
100 Constant Hun1
HEX
100 Constant Hun2
DECIMAL
CR ." Hun1= " Hun1 .
CR ." Hun2= " Hun2 .
```

3. Constants, variables, and the dictionary.

- (a) Execute:

```
hex
VARIABLE trythis 16161616 trythis !
VARIABLE andthis 32323232 andthis !
' trythis >name 7 - CONSTANT tryptr
' andthis >name 7 - CONSTANT andptr
tryptr U.
andptr U.
tryptr 16 - 60 dump
```

- (b) Look at your dump. Memory addresses are in the leftmost column. Following that are 16 groups of 2 hex digits each. These show the contents of 16 bytes of memory, in hex. The rightmost column shows the contents of the same 16 bytes in ASCII.

- (c) Each group of 4 bytes represents one unit of information or punctuation in FORTH. Figure out whether your units align neatly with the dump, that is, does each row of the dump have four complete groups on it? Or are they offset by 1, 2, or 3, bytes? If there is an offset, redefine `tryptr` so that every line of the dump has four complete groups of 4 bytes each.
- (d) Find the names `trythis`, `andthis`, `tryptr`, and `andptr` on your dump. If you cannot see `trythis`, start the dump at a lower memory address. Dump more bytes, if needed, to see the end of the dictionary for `andptr`.
- (e) Look at the number in the location just before or after the name. That is the number of letters in the name.
- (f) Capture and print the results of your last dump that shows both variable names and both constant names. Look at it. Can you find any patterns? Some implementations have markers before and after every symbol name. Does yours?

4. Code Field Addresses (CFAs).

Part of the dictionary entry for every symbol is the CFA, which is a kind of type tag. All constants have the same CFA tag value, variables and functions have different tag values. The FORTH interpreter uses the CFA to implement the correct semantics when it executes a program.

- (a) The single quote operator, called “tick”, suppresses semantic interpretation of the following word and leaves its CFA on the stack. Do this to learn the CFA of `VARIABLE` in your implementation:

```
hex
' trythis .
```

- (b) Now use the name of `tryptr` to discover the CFA for `CONSTANT`.
- (c) In my implementation, the four bytes following the name are the link field and the four bytes after that are the CFA, which should be the same for the two variables. In Win32 FORTH, the name is followed by a marker, the link field, and a pointer to a different area of memory, where the code and parameter fields (the storage object) are located. Remember that, with the Intel 32-bit architecture, the high-order byte is on the right, that is, the number 2, in memory, is four bytes: 02 00 00 00 and the address 00164326 looks like this: 26 43 16 00. Figure out how YOUR implementation works and make a diagram of how to find each of the fields of a dictionary entry (name field, code field, parameter field, and markers, if any).
- (d) After the code field of an object comes the parameter field: this is what holds the data for a constant or a variable or the byte-codes for a function. Find the values 16161616 and 32323232 on the dump. Find the address of `trythis` in the parameter field of `tryptr`.

5. Arrays.

`CREATE` makes the name, link field, and code field of a dictionary entry. The comma operator puts a value into the next free place in the dictionary.

- (a) This is how you can allocate and initialize an array. Execute:

```
decimal
CREATE array 2 , 5 , 17 , 34 , 64 , 65 , 66 ,
' array >name 7 - CONSTANT arrayptr
```

- (b) Do a complete dump of these two objects (two memory areas, if necessary.) Find the names `array` and `arrayptr` on your dump. Look at the byte count adjacent to the name. Find the code fields for these objects. Are they the same or different from those of `trythis` and `tryptr`?
- (c) Find the values 2 , 5 , 17 , 34 , 64 , 65 , and 66 in the parameter field of the array. Why are they hard to find at first?

6. Here we are.

Execute:

```
hex
VARIABLE another 10 allot
here CONSTANT endit
' another >name 7 - CONSTANT anotheradr
```

- (a) Do a complete dump of these two objects (two memory areas, if necessary.)
- (b) The constant named `endit` marks the end of the uninitialized array named `another`. How many bytes were allocated for `another`?
- (c) What is the address of `endit`? What is stored in `endit`?
- (d) What does `here` mean?