1. (1) 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 .
```

2. Constants, variables, and the dictionary.

(a) Execute:

```
hex
VARIABLE tryme 16161616 tryme!
VARIABLE andme 32323232 andme !
tryme 28 - CONSTANT tryhead
255 CONSTANT anotherconst
tryhead U.
tryhead 100 dump
```

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

If you are working in gForth, each group of 8 bytes represents one unit of information or punctuation in FORTH: a number or a word or a type tag. In Win32FORTH, each group of 4 bytes represents a unit.

(c) Names are variable length. Find the name `andme` in the rightmost column on your dump. Look at the hex codes for the information-unit just before the name. That is the number of letters in the name. Note that the least-significant byte of each number on the left end of the number in an Intel processor. The unit before that is a pointer to the beginning of the dictionary entry for `tryme`.

Figure out whether your units align neatly with the dump, that is, does each row of the dump have two (for gforth) or four (for Win32Forth) complete groups on it? Or are they offset by 1, 2, or more, bytes? If there is an offset, redefine `tryhead` with a larger constant so that every line of the dump has two (or four) complete groups of 8 bytes (or 4 bytes) each.

(d) (1) Adjust the constant value of `tryhead` until your dump starts two complete units before the beginning of the name “tryme”. These two units are a pointer to the previous dictionary entry and the number of letters in “tryme”.

If you cannot see the link field and the length field that precede `tryme`, start the dump at a lower memory address. Dump more bytes, if needed, to see the number 255 at
the end of the dictionary entry for anotherconst. This will look like 00 00 00 FF in hex. Capture the results of this dump in a .txt file. The file should show the complete dictionary entries for both variables and both constants.

(e) (2) Find the names tryme, andme, tryhead, and anotherconst on your dump. Highlight these names and the corresponding byte codes on your dump. Circle the numbers that give the length of each name.

(f) (2) Pointers. In gforth, the first unit in each dictionary entry is a pointer (called the link) to the beginning of the previous word in the dictionary. Highlight the pointer for andme and highlight the address of the beginning of tryme in the leftmost column of the dump. Note that the byte order is different in the two copies of this address.

Part of the dictionary entry for every symbol is the code field, which is a kind of type tag. All constants have the same type tag value, variables and functions have different tag values. Each programmer-defined data types has a unique type tag, plus another field that indicates it is a CREATEd type. The FORTH interpreter uses the code field to implement the correct semantics when it executes a program.

(a) (1) On your dump-printout, look at the byte codes that follow the two variable names names. You should find one field that is the same for the two variables: this is the code field, and it contains the type code for VARIABLE. Circle or highlight the type codes on your dump.

(b) (1) Now discover the code field tag for CONSTANT by looking in the same position after the names of the two constants. Circle or highlight the codes on your dump.

(c) (1) Data Fields. In gforth, the data field of an object comes after the code field; in Win32-FORTH, the data is in a different location and a pointer to that location follows the code field. The data field of an object 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. Highlight or circle these data fields on your dump.

(d) (1) Find the address of the beginning of your dump in the data field of tryhead. Highlight it.

4. Arrays.
CREATE makes the link field, name field, code field, and a and a one-unit data field for a dictionary entry. If more memory is needed, the rest of the data field must be created by using ALLOT or comma. (The comma operator moves a value from the stack into the next free place in the dictionary.

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

```
CREATE array 2 , 5 , 11 , 22 , 40 , 41 , 41 ,
array 28 - CONSTANT arraryptr
```

(b) (2) Do a complete dump of these two objects. Find and highlight the names array and arrayptr on your dump. Find and highlight the code fields for these objects. Are they the same or different from those of tryme and tryptr?

(c) Find the hex values 2 , 5 , 11 , 22 , 40 , 41 and 41 in the parameter field of the array. (Decimal values 2 , 5 , 17 , 34 , 64 , 65 , and 66.)
5. Here we are.

The Forth function `here` returns the address of the first unallocated byte in the dictionary. Immediately after declaring a new object, `here` can be used to find the address of the end of the object. Execute:

```
VARIABLE another 10 allot
here CONSTANT endit
another CONSTANT anotherref
```

(a) (1) Do a complete dump of these two objects. The constant named `endit` marks the end of the uninitialized array named `another`. The address of the first byte in `another`'s data field can be found by executing:

```
another .
```

How many bytes were allocated for the data field of `another`? Why?

(b) (1) What is the address of `endit`? What is stored in `anotherref`?

(c) (1) A pointer is stored in `endit`. What does it point at?