Wesleyan University, Fall 2021, COMP 211
Homework 2: More information Representation
Due by 11:59pm on September 21, 2021

---

1. Written problems (5 points)

Problem 1. *In the following, "number" means non-negative integer; representation, numeral, etc., always means 8-bit unsigned binary representation; and the division operator / means integer division (i.e., always round down, so 8/2 = 4 and 7/2 = 3).*

Suppose the numeral for the number n is $b_7b_6b_5b_4b_3b_2b_1b_0$.

(a) *What is the numeral for $n/2$?*
(b) *What is the numeral for $2n$? Assume that if the numeral requires more than 8 bits, then the most-significant bits are dropped to obtain an 8-bit numeral. In other words, if $n \geq 128$, then your answer will be a numeral that does not actually represent $2n$.*
(c) *What number is represented by $b_7 \ldots b_1 0$ (i.e., the same representation as n, except the least significant bit is 0)? Express your answer as some arithmetic function of n.*
(d) *What number is represented by $b_7 \ldots b_1 1$? Express your answer as some arithmetic function of n.*

For each of these problems, make use of the fact that the numeral for n is a listing of the coefficients when you express n as a sum of powers of 2. For example, we are given that $n = b_7 \cdot 2^7 + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$. Use this to write $n/2$ as a sum of powers of 2, which tells you the numeral that represents $n/2$.

Solution:

(a) Expressed as a sum of powers of 2, $n/2 = 0 \cdot 2^7 + b_7 \cdot 2^6 + \cdots + b_1 \cdot 2^0$, so the numeral for $n/2$ is $0b_7 \ldots b_1$.
(b) Expressed as a sum of powers of 2, $2n = b_7 \cdot 2^8 + b_6 \cdot 2^7 + \cdots + b_1 \cdot 2^2 + b_0 \cdot 2^1 + 0 \cdot 2^0$. Without overflow, the numeral would be $b_7 \ldots b_0 0$, which is 9 bits. We drop the most significant bit to get $b_6 \ldots b_0 0$
(c) The number represented by $b_7 \ldots b_1 0$ is $b_7 \cdot 2^7 + \cdots + b_1 \cdot 2^1 + 0 \cdot 2^0 = 2(n/2)$.
(d) The number represented by $b_7 \ldots b_1 1$ is $b_7 \cdot 2^7 + \cdots + b_1 \cdot 2^1 + 1 \cdot 2^0 = 2(n/2) + 1$.

Problem 2. *Consider the pseudocode program in Figure 1. Describe the evolution of the state for the entire execution of this program, starting from the empty state. To do this, you will have to describe how each line changes the state (if it does). So your description will probably be a sequence of sentences something like "After line X, the state is Y because Z." Use the "list of bindings" notation rather than table notation. Your description must explain each time the state changes.*

Solution: Here is the sequence of states:

(1) We start with the empty state $s_0 = \{\}$.
(2) A variable declaration adds an entry with no value, so after Lines 1 and 2, the state is $s_1 = \{i \mapsto ?, x \mapsto ?\}$.

```
1  int i
2  int x
3
4  i ← 4
5  x ← 3
6  while i < 7 do
7      x ← x + i
8      i ← i + 2
9  endw
```

FIGURE 1. A pseudocode program.

(3) The expressions on lines 4 and 5 do not depend on any variables, so they just set the corresponding bindings in the state: $s_2 = \{i \mapsto 4, x \mapsto 3\}$.
(4) We evaluate $i < 7$ with the state s_2; since $s_2(i) = 4 < 7$, the test evaluates to true, so we enter the loop body.
   (a) For line 7, $x + i = s_2(x) + s_2(i) = 3 + 4 = 7$, so we update the state to bind 7 to x: $s_3 = \{i \mapsto 4, x \mapsto 7\}$.
   (b) For line 8, $i + 2 = s_3(i) + 2 = 4 + 2 = 6$, so we update the state to bind 6 to i: $s_4 = \{i \mapsto 6, x \mapsto 7\}$.
(5) We evaluate $i < 7$ with the state s_4; since $s_4(i) = 6 < 7$, the test evaluates to true, so we enter the loop body. Using the same reasoning as before, at the end of the loop body the state is $s_5 = \{i \mapsto 8, x \mapsto 13\}$.
(6) We evaluate $i < 7$ with the state s_5; since $s_5(i) = 8 \not< 7$, the test evaluates to false, and so we do not enter the loop and end the program. Thus s_5 is the final state.

PROBLEM 3. *The goal of cryptography is to transform a message in such a way that even if an adversary sees the message, the adversary is unable to read or understand the message. The Caesar Cipher, attributed to Julius Caesar, some 2000 years ago, works by taking each letter of plaintext and substituting the letter that is k locations later in the alphabet. For example, if $k = 3$, as in the original Caesar Cipher, we would have the following mapping between plaintext letters and ciphertext letters.*

```
abcdefghijklmnopqlrstuvwxyz : plaintext letter
defghijklmnopqlrstuvwxyzabc : ciphertext letter
```

Using this mapping to encrypt the plaintext of "hello world" would give the ciphertext of "khoor zruog". Now since there are only 26 letters in the alphabet, the key value, k can take on at most 25 possible values: hence, this cipher is not very secure, since the time it would take to try all 25 key values is negligible.

Suppose a Caesar Cipher has been used to encrypt two lower case alphabet letters, producing two new lower case alphabet letters. The resulting letters are then encoded using ISO-8859-1 to produce the final ciphertext. The key used to do the encryption is $k = 20$: that is, each of the original letters has been mapped to the letter 20 places later in the 26-letter alphabet. If `ae` *were the original plaintext, then the ISO-8859-1 encodings of the letters* `uy` *would be the ciphertext output by the encryption process.*

```
$ gcc -o hw2a hw2a.c

$ ./hw2a
Enter non-negative decimal integer to convert: 10
Conversion to binary: 0000000000001010

$ ./hw2a
Enter non-negative decimal integer to convert: 32
Conversion to binary: 0000000000100000

$ ./hw2a
Enter non-negative decimal integer to convert: 23564356433
Conversion to binary: 1111111111111111
Error occurred
```

FIGURE 2. Some sample traces from `hw2a`.

Now, suppose the two bytes, 01101000 01101001*, are the output of the encryption process. What is the original plaintext?*

Solution: The two bytes represent `hi` when encrypted. To decrypt, we reverse shift each letter 20 places in the alphabet, obtaining the plaintext `no`. Alternatively, we can simply look up using the key mapping to see what `hi` maps to when encrypted.

2. CODING PROBLEMS (15 POINTS)

PROBLEM 4. *You will write a program to convert from decimal to binary. Your program will read in a non-negative integer entered by a user, and will print out the corresponding unsigned binary representation. To achieve this, there are multiple different solutions you may choose to implement. You may assume that the user will enter a non-negative integer (i.e., it does not matter what your program does if the user enters anything else). Your program should report an error (and also possibly an incorrect result) if the user enters a non-negative integer that requires more than* 16 *bits to represent.*

For this program, you may not use any library functions such as `pow`*. Additionally note that* `pow` *operates on numbers of floating point type whereas the user is entering a number of integer type. You should always use the most appropriate type for the information being represented.*

You should name your program `hw2a.c`*. Figure 2 is an example trace of the output that should be seen when your program is executed. As a reminder, the command-line for compiling your program is also shown, which compiles the source code* `hw2a.c` *to the executable program* `hw2a`*. Remember, to execute a program that is in the current working directory, you must use the command* `./<program-name>`*, where* `<program-name>` *is the name of the program (*`hw2a` *in this case). Because* `.` *is shorthand for "current working directory," this command says to find* `<program-name>` *in the current working directory; by default, the shell will* not *look in the current working directory for executable programs, so you have to tell it explicitly to do so!*

PROBLEM 5. *You will write a program to perform a Caesar shift on a single character. Have your program first read in the character to be shifted, and then read in the amount to shift by. Your program should then compute and output the shifted character. You should assume that the user*

```
$ ./hw2b
Enter lower-case letter to encrypt: d
Enter the shift amount for Caesar cipher: 5
Ciphertext is i

$ ./hw2b
Enter lower-case letter to encrypt: C
Error: user did not enter lower-case letter, exiting

$ ./hw2b
Enter lower-case letter to encrypt: q
Enter the shift amount for Caesar cipher: 32
Ciphertext is w
```

FIGURE 3. Some sample traces from `hw2b`.

enters a lower-case letter as the character: if this is not the case, your program should exit. Hint: a character is represented as 8-bit ASCII in C. You should name your program `hw2b.c`. *Figure 3 shows a few sample runs of the program.*

3. Going further

This question is *not* to be submitted, but is just a bit of food for thought. In Problem 4, you assumed that the largest non-negative integer that the user could enter could fit into 16 bits. This assumption simplified the calculations that your program had to perform. Modify your program so that this assumption is no longer needed.

Now something you may not have realized is that the number of bits used in a type such as an `int` can differ on different hardware implementations. Typically there is some guaranteed minimum number of bits that will be used to represent a a type; for instance in C, an `int` is guaranteed to be at least 16 bits, but could be larger (use more bits) depending on the hardware. This variability can be problematic: ideally we'd like our programs to be portable, able to be run on any hardware without needing to do something special. Additionally, for network and systems applications, such as when we are crafting a header for a network packet, or for performing cryptographic calculations on data, we need to be able to assume that a type uses exactly the same fixed number of bits, regardless of where it is run. To address this issue, C has a `stdint.h` library (include the same way that `stdio.h` is included). This library provides types that are guaranteed to only use the specified number of bits regardless of where they are run. For instance the `uint8_t` type provided by the library is guaranteed to be an unsigned integer using exactly 8 bits. Similarly an `int16_t type` is guaranteed to be a signed integer using exactly 16 bits. The base set of types provided are `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t`, and `uint32_t`. Going back to Problems 3 and 4 of this homework, think about how you might have chosen the types for your variables more precisely: for instance, how many bits does each variable really need to use?

4. Submission

Upload your written work as `hw2.pdf` and your code solutions as `hw2a.c` and `hw2b.c` to the Google Drive directory I have created for you named `comp211-f21-USERNAME/hw1/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.