15-122: Principles of Imperative Computation

Lab Week 1, Wednesday — Tom Cortina, Nivedita Chopra

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab-integers .
% cd lab-integers
```

You should write your code in a file, `reverse.c0`, in the directory `lab-integers`.

Grading: You will work through (1.a) and (1.b) as a group. Finish (1.c) for credit; additionally get (1.d) working for extra credit. You can work on these problems with your neighbors and compare notes and solutions!

Manipulating integers with a loop

For these two tasks, you'll need to use a loop to manipluate integers. We can identify two ways of manipulating integers in C0:

- The mathematical operations of multiplication (`a * b`), division (`a / b`), modulo (`a % b`) addition (`a + b`), subtraction (`a - b`), and negation (`-a`).
- The bitwise operations bitwise-and (`a & b`), bitwise-or (`a | b`), bitwise-xor (`a ^ b`), bitwise negation (`~a`), left shift (`a << b`) and right-shift (`a >> b`).

We don't always think about these operations as distinct categories! Sometimes, for instance, we think about `a << b` as the mathematical operation $a \times 2^b$. But for this assignment we will make the distinction.

(1.a) Our first task will be to reverse the digits in a seven-digit decimal number (a number with fewer digits will be treated as having leading zeros). There's more than one way to do this! The examples below show one way of reversing a number:

i	x	y
0	1234567	0
1	123456	7
2	12345	76
3	1234	765
4	123	7654
5	12	76543
6	1	765432
7	0	7654321

i	x	y
0	15122	0
1	1512	2
2	151	22
3	15	221
4	1	2215
5	0	22151
6	0	221510
7	0	2215100

i	x	y
0	2400000	0
1	240000	0
2	24000	0
3	2400	0
4	240	0
5	24	0
6	2	4
7	0	42

Can you suggest a couple of loop invariants for the algorithm above? **Hint:** you may want to use the `POW` specification from lecture. What can you say about `POW(10,i)`?

`\\@loop_invariant` _____

`\\@loop_invariant` _____

`\\@loop_invariant` _____

`\\@loop_invariant` _____

Rememeber that if you use `POW`, you need your loop invariants to also ensure that the exponent will always be nonnegative.

(1.b) Here's two example traces of a different algorithm for reversing a seven-digit number; this algorithm has more complicated invariants. Can you state some of them? Think about division and modulo.

i	j	x	y
10000000	1	1234567	0
1000000	10	1234560	7000000
100000	100	1234500	7600000
10000	1000	1234000	7650000
1000	10000	1230000	7654000
100	100000	1200000	7654300
10	1000000	1000000	7654320
1	10000000	0	7654321

i	j	x	y
10000000	1	15122	0
1000000	10	15120	2000000
100000	100	15100	2200000
10000	1000	15000	2210000
1000	10000	10000	2215000
100	100000	0	2215100
10	1000000	0	2215100
1	10000000	0	2215100

`\\@loop_invariant` _____

`\\@loop_invariant` _____

`\\@loop_invariant` _____

`\\@loop_invariant` _____

(1.c) Now you have two good sets of loop invariants: you can use either one of them to implement a function that reverses the decimal digits in a nonnegative number! (There are other ways, too.)

In `reverse.c0`, write a function `reverse_dec` that reverses the decimal digits in a nonnegative number with at most 7 decimal digits. Treat a number with fewer digits as if it has leading zeroes.

Use only *mathematical* operations on integers: `* / % + -`. You shouldn't have to use `POW` outside of contracts, but you can if you get stuck.

```
% coin -d reverse.c0
-->> reverse_dec(7654321);
1234567 (int)
-->> reverse_dec(1512200);
22151 (int)
-->> reverse_dec(42);
2400000 (int)
```

You can test your code against our test cases by running `cc0 -d -x reverse.c0 test-dec.c0`

(1.d) In `reverse.c0`, write a function `reverse_hex` that reverses all the hex digits of any integer:

Use only *bitwise* operations on integers: `& | ^ ~ << >>` to manipulate the input (you may need to manipulate a counter using mathematical operations). Don't use `POW` in your code.

```
% coin -d -lutil reverse.c0
-->> int2hex(reverse_hex(0x195D3B7F));
"F7B3D591" (string)
-->> int2hex(reverse_hex(0xC0CAFE));
"EFAC0C00" (string)
-->> int2hex(reverse_hex(16));
"01000000" (string)
-->> reverse_hex(int_min());
8 (int)
-->> reverse_hex(int_max());
-9 (int)
```

You can test your code against our test cases by running `cc0 -d -x reverse.c0 test-hex.c0`