Lecture Notes on Ints

15-122: Principles of Imperative Computation
Frank Pfenning, modified by Jamie Morgenstern

Lecture 3
January 18, 2011

1 Reasoning with Loop Invariants, part 2

Suppose we want to write a function, `fib`, which computes the nth Fibonacci. Recall the mathematical definition of the Fibonacci sequence:

$$\begin{aligned} F[0] &= 0 \\ F[1] &= 1 \\ F[k] &= F[k-1] + F[k-2] \end{aligned}$$

We can write a simple recursive function to compute the nth Fibonacci:

```
// Computes the nth Fibonacci
int fib(int n) {
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

This function looks like a truthful representation of the math we just wrote down. However, something is wrong with both our mathematical definition and the function. What happens if we call the function with -4?

What we intended when we wrote down our imprecise version of F was to say "for k at least 2"; e.g., that we we were only defining the sequence for nonnegative indices. In order to enforce this for our function, we need to write a precondition:

```
// Computes the nth Fibonacci
int fib(int n)
//@requires n >= 0;
{
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return fib(n - 1) + fib(n - 2);
   }
}
```

Now, this function will not have a bug when we call it with a negative number: *it is not defined for negative values*. Unfortunately, this version of our function is inefficient. When we call the function with a positive $k > 2$, how many recursive calls do we see made from the first level? The first call to `fib` calls `fib` two more times, each of which may call `fib` two more times. In general, these recursive calls are going to be made at k levels, since the arguments in the recursive calls are only one or two smaller than the initial call, and at each level, we see twice as many calls to `fib`. So, there are about 2^k calls to `fib` to calculate `fib(k)`! Can we do better?

What if we decided to instead compute the Fibonaccis starting at the 0th Fibonacci, and work our way up to k? We could then write a faster program if we only had to compute the ith Fibonacci once for each $i < k$. However, whenever we write a more effienct but less obvious implementation of a mathematical definition, we should take care to ensure our more efficient implementation actually implements the original definition. So, we will write a postcondition which will check that our result always matches with the original inefficient computation:

```
// Efficiently computes the nth Fibonacci
int fastfib(int n)
//@requires n >= 0;
//@ensures \result == fib(n);
{
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   }
  int k = 0;
  int curr = 0;
  int next = 1;
  while (k <= n) {
    int tmp = curr + next;
    curr = next;
    next = tmp;
    k++;
 }
 return curr;
}
```

Try to prove the postcondition using the precondition. The two base cases are fine, but what about the larger cases? If we ever write a loop, we should write loop invariants that help us prove our postcondition:

```
// Efficiently computes the nth Fibonacci
int fastfib(int n)
//@requires n >= 0;
//@ensures \result == fib(n);
{
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   }
  int k = 0;
  int curr = 0;
  int next = 1;
  while (k <= n)
  //@loop_invariant fib(k) == curr && fib(k + 1) == prev;
  {
    int tmp = curr + next;
    curr = next;
    next = tmp;
    k++;
 }
 return curr;
}
```

Let's prove the loop invariant. First, we have to show the loop invariant is satisfied before the loop is executed:

$k = 0, curr = 0, next = 1, n \geq 0$	Our Assumptions
fib(0) = 0	By Assumption $k = 0, curr = 0$, substitutions of 0 for k, and definition of fib
fib(1) = 1	By Assumption $k = 0, next = 1$, substitutions of 0 for k, and definition of fib

Now, we must assume that before we execute the body of the loop, the loop invariant holds and show that executing the body of the loop maintains the invariant:

fib(k) = curr, fib(k + 1) = next, $k \leq n, n \geq 0$	Our Assumptions
fib(k') = curr', fib(k' + 1) = next'	Need to Show
k' = k + 1, curr' = next, next' = curr + next	By the reassignments in the loop
fib(k') = fib(k + 1) = next = curr'	By our second assumption and the reassignment substitions
fib(k' + 1) = fib(k + 2)	By reassignment substitions
fib(k + 2) = fib(k) + fib(k + 1)	By definition of fib
fib(k' + 1) = curr + next	By the two previous lines and assumptions 1 and 2
fib(k' + 1) = next'	By the reassignment of curr and the previous line

Now, let's try to prove the postcondition from the precondition and the loop invariant. We have 3 separate cases, the two base cases and the final return statement. Base cases:

$k = 0, curr = 0, next = 1, n = 0$	Our Assumptions
fib(n) = fib(0) = 0	By Assumption $n = 0$, substitution of 0 for n, and definition of fib

$k = 0, curr = 0, next = 1, n = 1$	Our Assumptions
fib(n) = fib(1) = 1	By Assumption $n = 1$, substitution of 1 for n, and definition of fib

For the final return statement:

fib(k) = curr, fib(k + 1) = next, $k > n, n \geq 0$	Our Assumptions
curr = fib(n)	Need to Show
fib(k) = curr	Assumption 1
???	

But here we get stuck! Since $k > n$, $fib(n) \neq fib(k) = curr$: we have made a mistake! Our loop break condition must be fixed so that, once we stop executing the loop, $k = n$. So, replace the line

```
while (k <= n)
```

with

```
while (k < n)
```

Then, we can finish the proof as $k = n$ when we return $curr$.

2 Introduction to Ints

Two fundamental types in almost any programming language are booleans and integers. Booleans are comparatively straightforward: they have two possible values (`true` and `false`) and conditionals to test boolean values. We will return to their properties in a later lecture.

Integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are considerably more complex, because there are infinitely many of them. Because memory is finite, only a finite subrange of them can be represented in computers. In this lecture we discuss how integers are represented, how we can deal with the limited precision in the representation, and how various operations are defined on these representations.

3 Binary Representation of Natural Numbers

For the moment, we only consider the natural numbers $0, 1, 2, \ldots$ and we do not yet consider the problems of limited precision. Number notations have a *base* b. To write down numbers in base b we need b distinct *digits*. Each digit is multiplied by an increasing power of b, starting with b^0 at the right end. For example, in base 10 we have the ten digits 0–9 and the string `9380` represents the number $9 * 10^3 + 3 * 10^2 + 8 * 10^1 + 0 * 10^0$. We call numbers in base 10 *decimal numbers*. Unless it is clear from context that we are talking about a certain base, we use a subscript$_{[b]}$ to indicate a number in base b.

In computer systems, two bases are of particular importance. *Binary numbers* use base 2, with digits 0 and 1, and *hexadecimal numbers* (explained more below) use base 16, with digits 0–9 and A–F. Binary numbers are so important because the basic digits, 0 and 1, can be modeled inside the computer by two different voltages, usually "off" for 0 and "on" for 1. To find the number represented by a sequence of binary digits we multiply each digit by the appropriate power of 2 and add up the results. In general, the value of a bit sequence

$$b_{n-1} \ldots b_1 b_0\ {}_{[2]} = b_{n-1} 2^{n-1} + \cdots + b_1 2^1 + b_0 2^0 = \sum_{i=0}^{n-1} b_i 2^i$$

For example, $\texttt{10011}_{[2]}$ represents $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19$.

We can also calculate the value of a binary number in a nested way, exploiting Horner's rule for evaluating polynomials.

$$10011_{[2]} = (((1 * 2 + 0) * 2 + 0) * 2 + 1) * 2 + 1 = 19$$

In general, if we have an n-bit number with bits $b_{n-1} \ldots b_0$, we can calculate

$$(\cdots((b_{n-1} * 2 + b_{n-2}) * 2 + b_{n-3}) * 2 + \cdots + b_1) * 2 + b_0$$

For example, taking the binary number $\texttt{10010110}_{[2]}$ write the digits from most significant to least significant, calculating the cumulative value from left to right by writing it top to bottom.

$$\begin{array}{rcrcrcr} & & & & 1 & = & 1 \\ 1 & * & 2 & + & 0 & = & 2 \\ 2 & * & 2 & + & 0 & = & 4 \\ 4 & * & 2 & + & 1 & = & 9 \\ 9 & * & 2 & + & 0 & = & 18 \\ 18 & * & 2 & + & 1 & = & 37 \\ 37 & * & 2 & + & 1 & = & 75 \\ 75 & * & 2 & + & 0 & = & 150 \end{array}$$

Reversing this process allows us to convert a number into binary form. Here we start with the number and successively divide by two, calculating the remainder. At the end, the least significant bit is at the top.

For example, converting 198 to binary form would proceed as follows:

$$\begin{array}{rcrcrcr} 198 & = & 99 & * & 2 & + & 0 \\ 99 & = & 49 & * & 2 & + & 1 \\ 49 & = & 24 & * & 2 & + & 1 \\ 24 & = & 12 & * & 2 & + & 0 \\ 12 & = & 6 & * & 2 & + & 0 \\ 6 & = & 3 & * & 2 & + & 0 \\ 3 & = & 1 & * & 2 & + & 1 \\ 1 & = & 0 & * & 2 & + & 1 \end{array}$$

We read off the answer, from bottom to top, arriving at $\texttt{11000110}_{[2]}$.

4 Modular Arithmetic

Within a computer, there is a natural size of words that can be processed by single instructions. In early computers, the word size was typically 8 bits; now it is 32 or 64. In programming languages that are relatively close to machine instructions like C or C0, this means that the native type `int` of integers is limited to the size of machine words. In C0, we decided that the values of type `int` occupy 32 bits.

This is very easy to deal with for small numbers, because the more significant digits can simply be 0. According to the formula that yields their number value, these bits do not contribute to the overall value. But we

have to decide how to deal with large numbers, when operations such as addition or multiplication would yield numbers that are too big to fit into a fixed number of bits. One possibility would be to raise overflow exceptions. This is somewhat expensive (since the overflow condition must be explicitly detected), and has other negative consequences. For example, $(n+n)-n$ is no longer equal to $n+(n-n)$ because the former can overflow while the latter always yields n and does not overflow. Another possibility is to carry out arithmetic operations *modulo* the number of representable integers, which would be 2^{32} in the case of C0. We say that the machine implements *modular arithmetic*.

In higher-level languages, one would be more inclined to think of the type of `int` to be inhabited by integers of essentially unbounded size. This means that a value of this type would consist of a whole vector of machine words whose size may vary as computation proceeds. Basic operations such as addition no longer map directly onto machine instruction, but are implemented by small programs. Whether this overhead is acceptable depends on the application.

Returning to modular arithmetic, the idea is that any operation is carried out modulo 2^p for the precision p. Even when the modulus is not a power of two, many of the usual laws of arithmetic continue to hold, which makes it possible to write programs confidently without having to worry, for example, about whether to write $x + (y + z)$ or $(x + y) + z$. We have the following properties of the abstract algebraic class of *rings* which are shared between ordinary integers and integers modulo a fixed number n.

Commutativity of addition	$x + y = y + x$
Associativity of addition	$(x + y) + z = x + (y + z)$
Additive unit	$x + 0 = x$
Additive inverse	$x + (-x) = 0$
Cancellation	$-(-x) = x$
Commutativity of multiplication	$x * y = y * x$
Associativity of multiplication	$(x * y) * z = x * (y * z)$
Multiplicative unit	$x * 1 = x$
Distributivity	$x * (y + z) = x * y + x * z$
Annihilation	$x * 0 = 0$

Some of these laws, such as associativity and distributivity, do *not* hold for so-called *floating point* numbers that approximate real numbers. This significantly complicates the task of reasoning about programs with floating point numbers which we have therefore omitted from C0.

5 An Algorithm for Binary Addition

In the examples, we use arithmetic modulo 2^4, with 4-bit numbers. Addition proceeds from right to left, adding binary digits modulo 2, and using a carry if the result is 2 or greater. For example,

$$\begin{array}{cccccccc} & 1 & 0 & 1 & 1 & = & 11 & \\ + & 1 & 0_1 & 0_1 & 1 & = & 9 & \\ \hline (1) & 0 & 1 & 0 & 0 & = & 20 & = 4 \pmod{16} \end{array}$$

where we used a subscript to indicate a carry from the right. The final carry, shown in parentheses, is ignored, yielding the answer of 4 which is correct modulo 16.

This grade-school algorithm is quite easy to implement in software, but it is not suitable for a hardware implementation because it is too sequential. On 32 bit numbers the algorithm would go through 32 stages, for an operation which, ideally, we should be able to perform in one machine cycle. Modern hardware accomplishes this by using an algorithm where more of the work can be done in parallel.

6 Two's Complement Representation

So far, we have concentrated on the representation of natural numbers $0, 1, 2, \ldots$. In practice, of course, we would like to program with negative numbers. How do we define negative numbers? We define negative numbers as additive inverses: *$-x$ is the number y such that $x + y = 0$.* A crucial observation is that in modular arithmetic, additive inverses already exist! For example, $-1 = 15 \pmod{16}$ because $-1 + 16 = 15$. And $1 + 15 = 16 = 0 \pmod{16}$, so, indeed, 15 is the additive inverse of 1 modulo 16.

Similarly, $-2 = 14 \pmod{16}$, $-3 = 13 \pmod{16}$, etc. Writing out the equivalence classes of numbers modulo 16 together with their binary

representation, we have

$\ldots$	-16	0	16	$\ldots$ 0000
$\ldots$	-15	1	17	$\ldots$ 0001
$\ldots$	-14	2	18	$\ldots$ 0010
$\ldots$	-13	3	19	$\ldots$ 0011
$\ldots$	-12	4	20	$\ldots$ 0100
$\ldots$	-11	5	21	$\ldots$ 0101
$\ldots$	-10	6	22	$\ldots$ 0110
$\ldots$	-9	7	23	$\ldots$ 0111
$\ldots$	-8	8	24	$\ldots$ 1000
$\ldots$	-7	9	25	$\ldots$ 1001
$\ldots$	-6	10	26	$\ldots$ 1010
$\ldots$	-5	11	27	$\ldots$ 1011
$\ldots$	-4	12	28	$\ldots$ 1100
$\ldots$	-3	13	29	$\ldots$ 1101
$\ldots$	-2	14	30	$\ldots$ 1110
$\ldots$	-1	15	31	$\ldots$ 1111

At this point we just have to decide which numbers we interpret as positive and which as negative. We would like to have an equal number of positive and negative numbers, where we include 0 among the positive ones. From this considerations we can see that $0, \ldots, 7$ should be positive and $-8, \ldots, -1$ should be negative and that the highest bit of the 4-bit binary representation tells us if the number is positive or negative.

Just for verification, let's check that $7 + (-7) = 0 \pmod{16}$:

	0	1	1	1
$+$	1_1	0_1	0_1	1
(1)	0	0	0	0

It is easy to see that we can obtain $-x$ from x on the bit representation by first complementing all the bits and then adding 1. In fact, the addition of x with its bitwise complement (written $\sim x$) always consists of all 1's, because in each position we have a 0 and a 1, and no carries at all. Adding one to the number $11\ldots11$ will always result in $00\ldots00$, with a final carry of 1 that is ignored.

These considerations also show that, regardless of the number of bits, -1 is always represented as a string of 1's.

In 4-bit numbers, the maximal positive number is 7 and the minimal negative numbers is -8, thus spanning a range of $16 = 2^4$ numbers. In

general, in a representation with p bits, the positive numbers go from 0 to $2^{p-1} - 1$ and the negative numbers from -2^{p-1} to -1. It is remarkable that because of the origin of this representation in modular arithmetic, the "usual" bit-level algorithms for addition and multiplication can ignore that some numbers are interpreted as positive and others as negative and still yield the correct answer modulo 2^p.

However, for comparisons, division, and modulus operations the sign does matter. We discuss division below in Section 9. For comparisons, we just have to properly take into account the highest bit because, say, $-1 = 15 \pmod{16}$, but $-1 < 0$ and $0 < 15$.

7 Hexadecimal Notation

In C0, we use 32 bit integers. Writing these numbers out in decimal notation is certainly feasible, but sometimes awkward since the bit pattern of the representation is not easy to discern. Binary notation is rather expansive (using 32 bits for one number) and therefore difficult to work with. A good compromise is found in *hexadecimal notation*, which is a representation in base 16 with the sixteen digits 0–9 and A–F. "Hexadecimal" is often abbreviated as "hex". In the concrete syntax of C0 and C, hexadecimal numbers are preceded by `0x` in order to distinguish them from decimal numbers.

binary	hex	decimal
0000	`0x0`	0
0001	`0x1`	1
0010	`0x2`	2
0011	`0x3`	3
0100	`0x4`	4
0101	`0x5`	5
0110	`0x6`	6
0111	`0x7`	7
1000	`0x8`	8
1001	`0x9`	9
1010	`0xA`	10
1011	`0xB`	11
1100	`0xC`	12
1101	`0xD`	13
1110	`0xE`	14
1111	`0xF`	15

Hexadecimal notation is convenient because most common word sizes (8 bits, 16 bits, 32 bits, and 64 bits) are multiples of 4. For example, a 32 bit number can be represent by eight hexadecimal digits. We can even do a limited amount on arithmetic on them, once we get used to calculating modulo 16. Mostly, though, we use hexadecimal notation when we use bitwise operations rather than arithmetic operations.

8 Bitwise Operations on Ints

Ints are also used to represent other kinds of data. An example, explored in the first programming assignment, is colors (see Section 11). The so-called ARGB color model divides an int into four 8-bit quantities. The highest 8 bits represent the opaqueness of the color against its background, while the lower 24 bits represent the intensity of the red, green and blue components of a color. Manipulating this representation with addition and multiplication is quite unnatural; instead we usually use bitwise operations.

The bitwise operations are defined by their action on a single bit and then applied in parallel to a whole word. The tables below define the meaning of *bitwise and* `&`, *bitwise exclusive or* `^` and *bitwise or* `|`. We also have *bitwise negation* `~` as a unary operation.

And

&	0	1
0	0	0
1	0	1

Exclusive Or

^	0	1
0	0	1
1	1	0

Or

|	0	1
0	0	1
1	1	1

Negation

~	0	1
	1	0

9 Integer Division and Modulus

The division and modulus operators on integers are somewhat special. As a multiplicative inverse, division is not always defined, so we adopt a different definition. We write x/y for *integer division* of x by y and $x\%y$ for *integer modulus*. The two operations must satisfy the property

$$(x/y) * y + (x\%y) = x$$

so that $x\%y$ is like the remainder of division. The above is not yet sufficient to define the two operations. In addition we say $0 \leq |x\%y| < |y|$. Still, this leaves open the possibility that the modulus is positive or negative when y does not divide x. We fix this by stipulating that integer division truncates

its result towards zero. This means that the modulus must be negative if x is negative and there is a remainder, and it must be positive if x is positive.

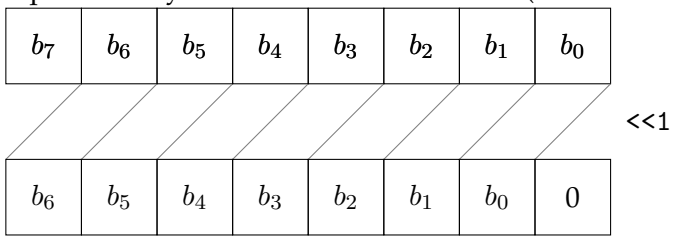
By contrast, the *quotient* operation always truncates down (towards $-\infty$), which means that the *remainder* is always positive. There are no primitive operators for quotient and remainder, but they can be implemented with the ones at hand.

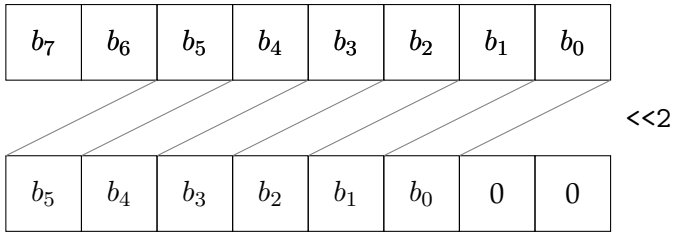
Of course, the above constraints are impossible to satisfy when $y = 0$, because $0 \leq |x\%0| < |0|$ is impossible. But division by zero is defined to raise an error.

10 Shifts

We also have some hybrid operators on ints, somewhere between bit-level and arithmetic. These are the *shift* operators. We write `x << k` for the result of shifting x by k bits to the left, and `x >> k` for the result of shifting x by k bits to the right. In both cases, k is masked to 5 bits before the operation is applied, which can be written as `k & 0x1F`, so that the actual shift quantity s has the property $0 \leq s < 32$. We assume below that k is in that range.

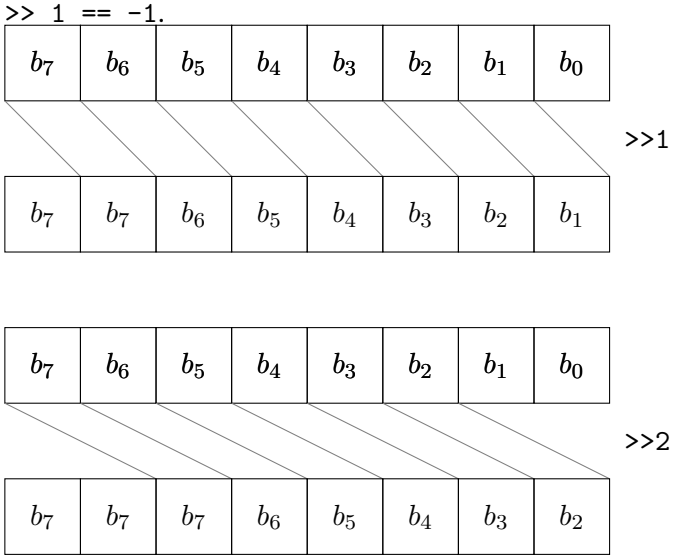
The left shift, `x << k` (for $0 \leq k < 32$), fills the result with zeroes on the right, so that bits $0, \ldots, k-1$ will be 0. Every left shift corresponds to a multiplication by 2 so `x << k` returns $x * 2^k$ (modulo 2^{32}).





The right shift, `x >> k` (for $0 \leq k < 32$), copies the highest bit while shifting to the right, so that bits $31, \ldots, 32-k$ of the result will be equal to the highest bit of x. If viewing x a an integer, this means that the sign of the

result is equal to the sign of x, and shifting x right by k bits correspond to integer division by 2^k except that it truncates towards $-\infty$. For example, `-1 >> 1 == -1`.



11 Representing Colors

As a small example of using the bitwise interpretation of ints, we consider colors. Colors are decomposed into their primary components red, green, and blue; the intensity of each uses 8 bits and therefore varies between 0 and 255 (or `0x00` and `0xFF`). We also have the so-called α*-channel* which indicates how opaque the color is when superimposed over its background. Here, `0xFF` indicates completely opaque, and `0x00` completely transparent.

For example, to extract the intensity of the red color in a given pixel p, we could compute `(p >> 16) & 0xFF`. The first shift moves the red color value into the bits 0–7; the bitwise and masks out all the other bits by setting them to 0. The result will always be in the desired range, from 0–255.

Conversely, if we want to set the intensity of green of the pixel p to the value of g (assuming we already have $0 \leq g \leq 255$), we can compute `(p & 0xFFFF00FF) | (g << 8)`. This works by first setting the green intensity to 0, while keep everything else the same, and then combining it with the value of g, shifted to the right position in the word.

For more on color values and some examples, see Assignment 1.

Notice that, colors are just integers for the computer. The machine does not distinguish between memory cells that represent colors and memory

cells that represent integers. Both are just bit patterns in memory and it is up to our interpretation what they represent. Ultimately, the program decides what it does with the integers. Does the program print it out as an decimal number? Then the bit pattern represents an integer. Does the program plot a pixel of a color corresponding to that bit pattern on the screen? Well, then the bit pattern represents a color. In either case, programs are free to use all arithmetic operations on the data. Since it is still useful for the human to know what a particular variable represents, C0 allows you to define types that you can use to document what representation makes sense for a variable. For instance, after you declare `pixel` to be an alias for the type `int` via

```
typedef int pixel;
```

you can declare a variable using `pixel c;` when you want to understand it as a color code in the rest of your program.

It is up to the programmer to define what the data means and how it is going to be interpreted. Its the programmer's responsibility to make sure that they are interpreted in a consistent way. C0 helps the programmer with this. For example, C0 distinguishes the type `bool` from the type `int`, because it is not clear which truth-value (true/false) to assign to a number.

Exercises

Exercise 1 *Write functions* `quot` *and* `rem` *that calculate quotient and remainder as explained in Section 9. Your functions should have the property that*

```
quot(x,y)*y + rem(x,y) == x;
```

for all ints x and y unless `quot` *overflows. How is that possible?*

Exercise 2 *Write a function* `int2hex` *that returns a string containing the hexadecimal representation of a given integer as a string. Your function should have prototype*

```
string int2hex(int x);
```

Exercise 3 *Write a function* `lsr` *(logical shift right), which is like right shift (*`>>`*) except that it fills the most significant bits with zeroes instead of copying the sign bit. Explain what* `lsr(x,1)` *means on integers in two's complement representation.*