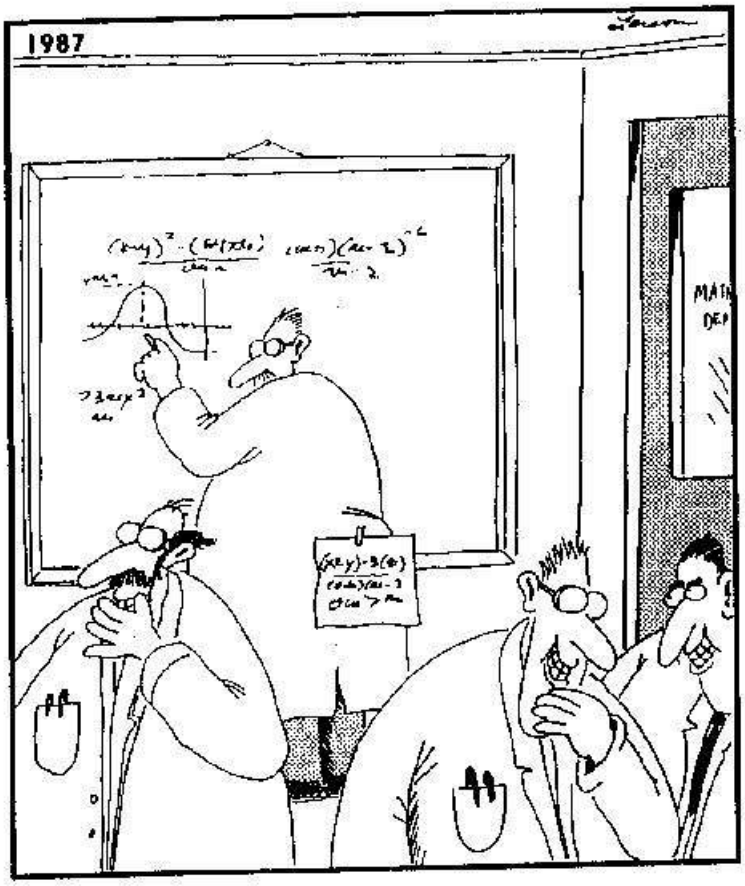
CS52 - Assignment 3

Due Monday 2/15 at 11:59pm



http://home.adelphi.edu/~stemkoski/mathematrix/comics.html

In class, we've been talking about how to develop "integers" that can be arbitrarily large. In this assignments, you will be working to flush out this functionality and create a package that supports many of the math operations you'd expect to have using this representation.

Starter

To help you get started, I have provided you with two files at:

`http://www.cs.pomona.edu/classes/cs52/assignments/assign3/`

- `assign3-starter.sml`: Download this file into your `cs052` directory and rename it to `assign3.sml`. This file includes some code to get you started as well as comments delimiting where various assignment parts below should go. If you're starting early (good work!), there may be some things at the top that we haven't talked about yet, but feel free to ignore those things for now (in particular, the `cs52int` datatype declaration).
- `assign3-tests.sml`: To help give you some examples of how to use some of the function you are writing and to give you *some* test examples, I've included a set of *basic* tests. These test are not comprehensive, but should help you test and debug your code. To run the tests first "`use`" your assignment solution and then run the tests by "`use`"ing the test file.

In addition to these resources, I have also uploaded a sample solution to problems 7-9 (the `toDigitList`, `fromDigitList`, `addDigitList` and `multDigitList`) onto the course's *sakai* website in the "Resources" section. If you had trouble with these on assignment 2, you might find these useful to help you craft some of your solutions for this assignment.

Advice

This assignment is not more difficult than previous ones, but it is longer—by a factor of three or four. Start early and build one part at a time, as outlined below. You are permitted to use any code given in class or in this document.

Read through this entire handout at least once before starting. Some of it may not make complete sense, but it will be helpful to see what you will be implementing.

I've tried to give you some hints throughout this assignment to help make your life easier. Think about each of the hints! I promise they will make your life easier.

The Setup

In class, we have been discussing arithmetic operations on lists of digits. These functions mimic the common "elementary school" algorithms on numbers in base 10. To make life more interesting (and to be able to support *really* large numbers) we are now making a very small change: from base 10 to another base. Our "digits" are numbers in the range from 0 through `base` $-$ 1, where `base` satisfies two conditions.

- `base` is greater than 1.

- `base*base` is small enough to be an ordinary SML `int`.

In the current implementations of SML, the largest int that can be stored is 107,3741,823, so this limits the base to be at most 32,767. We normally take `base` to be a power of two and will generally use the largest possible power of two, 2^{14} or $16,384$, but your code should work for any of the possible values.

Like we have been doing in class (and on the last assignment), we will write our digit lists so that the least significant "digit" is at the front of the list, and lists will be *normalized,* meaning that they are free of trailing zeroes.

A few quick examples:

- If `base = 4` then the digit list $[3, 2, 1]$ represents the "number" $3 * 4^0 + 2 * 4^1 + 1 * 4^2 = 15$.
- If `base = 9` then the digit list $[7, 4, 0, 2]$ represents the "number" $7*9^0+4*9^1+0*9^2+2*9^3 = 1,501$
- If `base = 16,384` then the digit list $[0, 7, 400]$ represents the "number" $0*16384^0+7*16384^1+4*16384^2 = 1,073,856,512$ (a pretty big number for just three small digits)

In your code, you should use the variable `base`, which is a global variable that will hold the base that you're working in. The default (i.e. what it's currently set at) is 16,384. I encourage you to adjust it to something more intuitive (like 2, 4, 16 or even 10 in some cases, though don't only test in base 10 because it is error prone since many of your base functions were already written assuming base 10) for testing your functions, however, make sure to change it back to the default before you submit your assignment.

Optional, but worth it

There are over 50 of you in the class and I am terrible with names, but I'm not bad with faces :) and I'll try to learn names. To help me with this, I'm strongly encouraging everyone who hasn't already to poke your head in my office sometime (office hours work great, but I'm around a lot outside of them), introduce yourself and tell me one interesting thing about you (e.g. "hey, I like to unicycle too!").

http://theoatmeal.com/pl/brain/name

The Fun!

1. **[7 points]** Deja Vu

   To get warmed up, we'll reimplement some of the functions you've done already in base 10, but now in *any* base, along with some additional support functions.

   Write the following functions in the appropriate section in your file. You may assume that the input lists are positive.

   For all of these problems *you must process the input lists in order*, i.e. you cannot reverse the list and process it in that order.

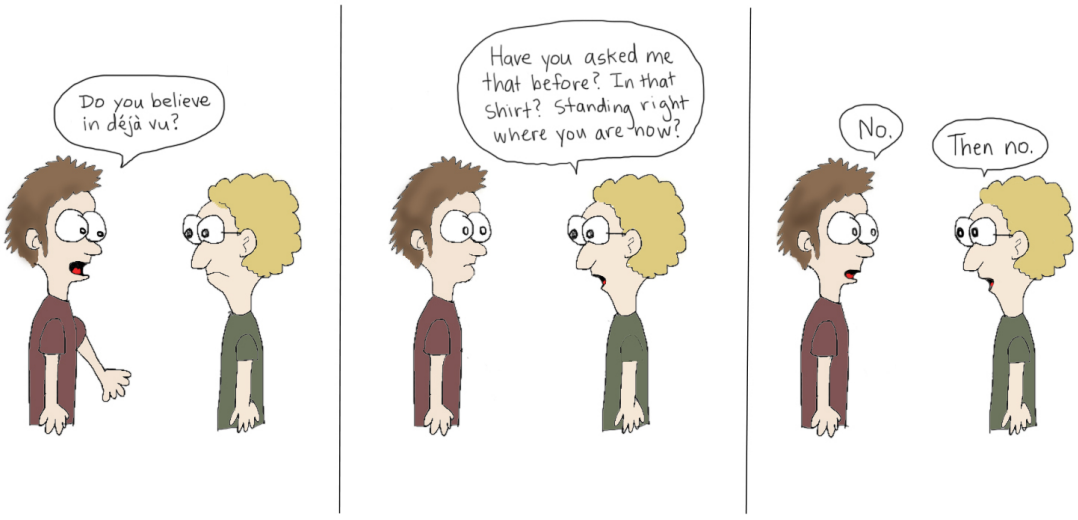
   (a) **[2 points]** `normalize: int list -> int list`
   Returns a copy of the argument list, without trailing zeroes. This is equivalent to normalizing 0047 as 47 for normal numbers. Remember that we're writing the lists with the least significant digits at the front, so you want to remove zeros at the *end* of the list.

   (b) **[1 point]** `addAsLists: int -> int list -> int list -> int list`
   Returns the sum of the two lists. Note the function has three parameters (we'll put a wrapper around it later). The first is the "carry-in", which for the top level call should be 0.

   (c) **[1 point]** `subtractAsLists: int -> int list -> int list -> int list`
   Returns the difference of the two lists, the first minus the second. Again, note that there are three parameters. The first argument is the "borrowed", which for the top level call should be 0. You may assume that the result will be non-negative.

   (d) **[2 points]** `compareAsLists: order -> int list -> int list -> order`
   Returns the order—`LESS`, `EQUAL`, or `GREATER`—of the two argument lists. The first argument is the "result so far" (similar to the "carry-in" or "borrowed" bit from the previous two functions). It will start as `EQUAL` for the top level call.

   (e) **[1 point]** `multiplyAsLists: int list -> int list -> int list`
   Returns the product of the two lists.



2. **[3 points]** Now we're getting somewhere

   Now that we have most of the functions to support math operations on our numbered lists, we're going to wrap this functionality into a proper datatype. At the top of the starter file we've declared the `cs52int`. This datatype allows us to add negative numbers (and zero) in a reasonable way (remember all our functions have assumed positive numbers) and to provide more usable functions than our list variants.

   The lists stored inside the `cs52int` datatype must meet the following constraints:

   - Each list is non-empty.
   - Each list contains integers between 0 and `base` − 1, inclusive.
   - No list contains trailing zeros.
   - The elements of the list are arranged from the least significant to the most significant. The least significant value is at the head of the list.

   Assuming you did everything correctly in the previous section, this should be easy to satisfy. One consequence of these restrictions is that representations are unique and values of type `cs52int` can be compared for equality.

   Write the functions described below that provide functionality for `cs52int`s. Use pattern-matching against the datatype `cs52int`. Although there may be several patterns in each function (for handling the three different types within the `cs52int`), no individual pattern will be complicated. All the hard work is passed to the list functions that you have already written!

   (a) **[0.5 point]** `negate: cs52int -> cs52int`
   Takes a `cs52int` k and returns $-k$.

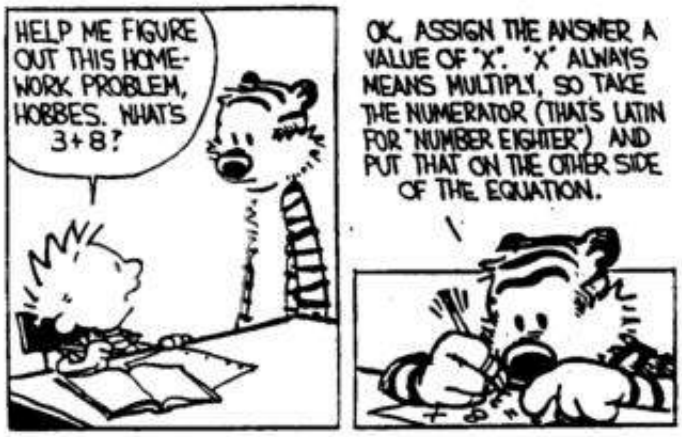
   (b) **[1 point]** `sum: cs52int -> cs52int -> cs52int`
   Takes two `cs52int`'s and returns their sum. Remember that `subtractAsLists` expects arguments which will produce a non-negative difference.

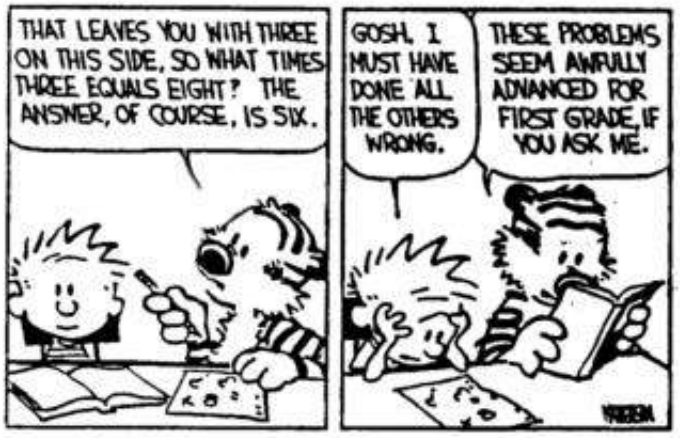
   (c) **[0.5 point]** `diff: cs52int -> cs52int -> cs52int`
   Takes two `cs52int`'s and returns their difference, the first minus the second.

   (d) **[0.5 point]** `prod: cs52int -> cs52int -> cs52int`
   Takes two `cs52int`'s and returns the product.

   (e) **[0.5 point** `compare: cs52int -> cs52int -> order`
   Takes two `cs52int`'s and returns their relationship as a value of type `order`. The type `order` is built-in and has three values: `LESS`, `EQUAL`, and `GREATER`. A value of `LESS` means that the first argument is less than the second.

3. **[1 point]** Write value declarations of type `cs52int` for the constants `zero`, `one`, and `two`. Be careful when declaring two since "`[2]`" is not a valid number list if the base is 2. These can be useful for testing your functions, though do test other values as well.





http://home.adelphi.edu/~stemkoski/mathematrix/comics.html

4. **[3 points]** More functionality!

   Now that we have the basic operations from part 2, we're going to add additional functionality. Write the functions below. Note that all of these should be *very* short and should utilize the functions from part 2.

   If you want to see some examples of the above functions being used, take a look at the test file that I made available as part of the starter.

   (a) **[1 point]** Implement functions that support the comparison operators:

   - `less: cs52int -> cs52int -> bool`
     Returns `true` if the first argument is strictly less than the second.
   - `lesseq: cs52int -> cs52int -> bool`
     Returns `true` if the first argument is not greater than the second.
   - `greater: cs52int -> cs52int -> bool`
     Returns `true` if the first argument is strictly greater than the second.
   - `greatereq: cs52int -> cs52int -> bool`
     Returns `true` if the first argument is not less than the second.
   - `unequal: cs52int -> cs52int -> bool`
     Returns `true` if the two arguments are different.

   (b) **[1 point]** Implement functions for the following math operations:

   - `abs: cs52int -> cs52int`
     Takes a `cs52int` k and returns its absolute value $|k|$.
   - `min: cs52int -> cs52int -> cs52int`
     Returns the lesser of its two arguments.
   - `max: cs52int -> cs52int -> cs52int`
     Returns the greater of its two arguments.

   (c) **[1 point]** And a few more just for fun (are we having fun yet?):

   - `sign: cs52int -> int`
     Returns -1, 0, or $+1$, according to whether the argument is negative, zero, or positive, respectively.
   - `sameSign: cs52int -> cs52int -> bool`
     Returns `true` if its two arguments are both negative, both zero, or both positive.
   - `square: cs52int -> cs52int` takes a `cs52int`
     k and returns its square k^2.

5. [**4.5 points**] A house divided cannot stand

   A keen observer will notice that we have overlooked an important operation...division! We could have done that like the other mathematical operations by manipulating lists. It turns out, though, that we can implement division on `cs52int`s using the other math operators we already have (like `sum`, `diff`, `prod`, etc.).

   Some terminology: the top number in division is called the *dividend* and the bottom number the *divisor*, e.g. in 11/2, 11 is the dividend and 2 is the divisor. The answer, is called the quotient, e.g. for 11/2, 5 is the quotient with a remainder of 1.

   (a) [**1 point**] One way to do division is to repeatedly subtract the divisor from the dividend and count how many times you can do it before the number becomes negative. For example, to figure out 11/2, note that you can subtract 2 fives times from 11 before it becomes negative.

   Implement a function called `slowDivide: cs52int -> cs52int -> cs52int` that implements division over `cs52int`s where the first input is the dividend and the second the divisor. The function should raise an exception called `CS52Div` if dividing by zero.

   Hints:

   - Don't overthink this function. Utilize our infrastructure (i.e. functions) that we have so far to build that operate on `cs52int`s to make life easy.
   - You'll need to handle different cases for positive and negative inputs, but you can pose most of the cases as other patterns, avoiding repeated code, e.g.:
     ```
     slowDivide (Neg x) (Neg y) = slowDiv (Pos x) (Pos y)
     ```
     (dividing two negative numbers is the same as dividing the numbers as if they were two positive numbers). In particular, only one of the patterns of this function should have any major substance to it.

   (b) [**3 points**] The reason this is called slow division is that it is slow, in particular, the run-time (i.e. speed) is dependent on the size of the quotient. If the quotient is large this can be very slow (remember that our numbers can get very, very big, particularly with large bases).

   We can generalize the previous approach as follows (written in pseudocode):

   ```
   input: dividend, divisor

   quotient = 0

   while dividend > divisor:
       k = pick some number where: dividend <= k*divisor
       dividend = dividend - k*divisor
       quotient = quotient + k

   return quotient
   ```

   Note that in SML, this loop is accomplished via recursion.

   The question is how do we pick k? In the slow algorithm, we always pick $k = 1$, but this algorithm will work for many other mechanisms for choosing k.

A choice that is much, much faster is to choose k to be the largest power of 2 such that $dividend \geq k * divisor$.

For example, if we wanted to figure out what 103/3 is:

- $k = 32$ the first time, since 3*32= 96 and 3*64 (the next larger power) is too large.
- Subtracting 96 from 103, we get 7.
- Now we want to figure out what 7/3 is. $k = 2$ now, since 3*2 = 6 and 3*4 is too large.
- Subtracting 6 from 7, we get 1. $k = 0$ now, since 3*1 is too large and we're done.
- Our final answer is: $32 + 2 = 34$ (which is the right answer).

Notice that this problem is very naturally recursive! Try a few on your own if you still don't see the recursion.

Write a function:

```
divide: cs52int -> cs52int -> cs52int
```

that implements this approach. Note you should only be searching powers of two. Do not try and search all numbers and then check if they're powers of two (this will be very slow again!). The function should raise an exception called `CS52Div` if dividing by zero.

Advice on how to tackle it:

- Write a helper function that calculates powers of 2, e.g. something like `power2 = fn:  cs52in -> cs52int`.
- Write another helper function that figures out the the largest power of 2 that when multiplied by the divisor is less than a specified value.
- Using the helper function, calculate the quotient. This function will be recursive.

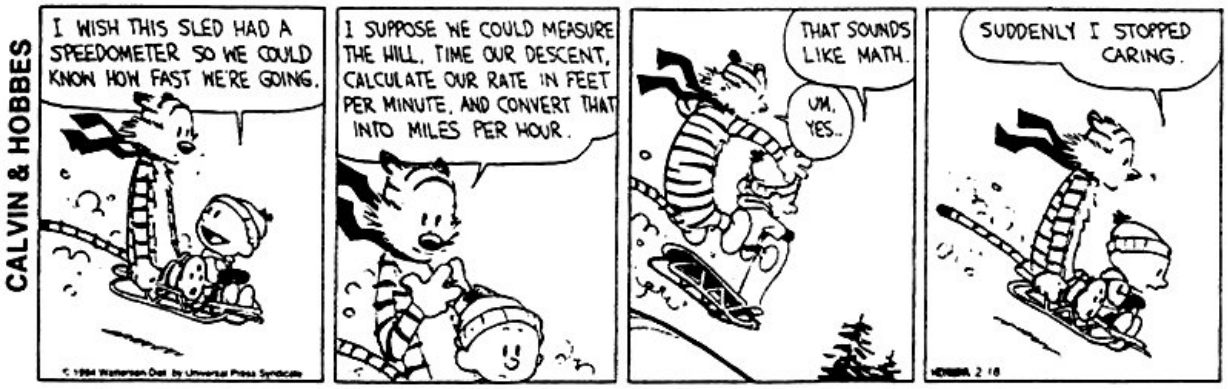
An aside: why is the second version faster? (You don't have to answer this, but think about it.) You can try it out by making calls to both versions where the quotient is very large and you should see the difference.

(c) **[0.5]** If we have division, we also should have remainder. Write a function

```
remainder: cs52int -> cs52int -> cs52int
```

that calculates the remainder when dividing the first number by the second. This should be very easy now that you've done division!

6. [**4.5 points**] The end is in sight!



`https://mrart.wikispaces.com/Math+Cartoons`

For the last part we are going to implement a few functions that will help make our life easier for displaying the numbers and in converting between `int`s and `cs52int`s.

(a) [**1 point**] `toInt: cs52int -> int`
Takes a `cs52int` and returns the corresponding `int`. It raises the built-in exception `Overflow` if the value will not fit in an ordinary integer. (Note that the system will automatically raise the exception when a value is out of range; you do not have to program it explicitly.)

(b) [**1 point**] `fromInt: int -> cs52int`
Takes an ordinary `int` and returns a `cs52int` with the same numerical value.

(c) [**2.5 points** `toString: cs52int -> string`
Takes a `cs52int` and returns a string with the decimal representation of the value. The resulting string will consist solely of digits, with a leading `~` for negative values.
Hints:

- The key idea is to calculate an output digit (0 through 9) at a time using some of your previous math functions.
- To calculate the `string` representation of a digit, you can do the following:
  `chr (ord #"0" + digit)`
  which works by using `ord` to get the number associated with "0" and then adds to it the digit and finally converts it back to the character representation. This works because 0-9 are given sequential numbers in ASCII.
- Don't forget that "^" is used to concatenate two strings.

(d) `fromString: string -> cs52int option`, is already written for you, but make sure to uncomment it. This can be useful for debugging.

7. All done!

Don't forget to set the base back to 16,384 before submitting.

When you're done

Double check the following things:

- Make sure your code runs without any errors (i.e. `use ''assign3.sml''` does not execute an error). If you didn't finish a function and it gives an error, just comment it out and leave a note.
- Make sure that your functions match the specifications *exactly*, i.e. the names should be exactly as written (including casing) and make sure your function takes the appropriate number of parameters and is curried/uncurried appropriately.
- Make sure you have used proper style and formatting. See the course readings for more information on this. Be informative and consistent with your formatting!
- Make sure you've properly commented your code. You should include:
  - A comment header at the top of the file with your name, the date, the assignment number, etc.
  - Each problem should be delimited by comment stating the problem number.
  - Each function should have a comment above it explaining what the function does.
  - Complicating or unusual lines in functions should also be commented.

  Don't go overboard with commenting, but do be conscientious about it.

When you're ready to submit, upload your assignment via the online submission mechanism. You may submit as many times as you'd like up until the deadline. We will only grade the most recent submission.

Grading

functions	23
comments/style	3
Total	26