Bignums Hints

CS51
Melissa Kwan

1 Extended Bignums Hints

This document provides a few conceptual as well as design hints for Bignums. Most people think that Bignums is the most difficult problem set of the year, so it's downhill from here :)

1.1 Exercise-specific tips

1. `less / greater / equal`
   (a) Think about the role of both negatives and the lengths of the respective bignums. You shouldn't need to define a condition for each one.
   (b) You only need to examine the elements of the list if they're the same length and have the same negation.
   (c) Can you define some of these in terms of the others?
   (d) If you had to copy and paste any significant chunk of code to write this function, think about how you can optimize it. You can write a clean solution in well under 10 lines.
2. `from_int, to_int`
   (a) `from_int`: Write out integers in terms of powers of `cBASE`. How can you use `mod` and `/` to break up the bignum into the "least significant" digits and the rest of the digits?
   (b) `to_int`: You know that if a bignum exceeds the maximum integer, `to_int` should output `None`. How can you use previously defined functions to check if this is the case?
3. `plus`
   (a) Think about what happens when you add a positive and negative number. How can you determine the sign?
   (b) The staff has implemented `plus_pos` for you, but it assumes that the sign of the resulting sum is positive. To maintain this invariant, you should make sure that if you call `plus_pos`, the sum is guaranteed to be positive.
   (c) Now, what will you do if the true sum is actually negative?
4. `times`
   (a) Again, think about powers of `cBASE`. It's helpful to look at an example of the grade-school algorithm for multiplication. The only thing that changed is that `cBASE` is no longer 10; it's 1000. How can you use your intuition from grade school to solve this problem in a similar way?
   (b) In the grade school algorithm, multiplying a n-digit number by a m-digit takes $n \cdot m$ multiplications of digits, added together. Is there a natural way you could break these $n \cdot m$ operations into chunks? (Don't be afraid of helper functions for this one.)
   (c) `1,234,057 * 1000 = 1,234,057,000` is equivalent to `[1; 234; 057] * 1000 = [1; 234; 057; 0]` for `cBASE = 1000`.
   (d) What's the sign when you multiply by 0?

1.2 General tips

1. Make sure you're not spelling out repetitive operations within functions; define those as internal helper functions.
2. Consider whether your helper functions are used outside of one function. If it's a pretty short helper function and it's only used in one place, you're better off defining the function within the main function.
3. Only specify the types of outer functions — no need to specify the types of inner helper functions.
4. Make sure your variables have descriptive names (don't name helper functions `help1, help2,` etc.). A good rule of thumb: your ideal variable name is 1-5 characters, and your ideal function name is 4-15 characters.
5. Use field punning to deconstruct records in the input when you can.

1.3 Testing: Utop Alternatives

1. `#use`. Change directories (`cd`) into your lab / problem set folder. Then, from `utop`, you can use `#use "mapfold.ml"` to directly import all your functions.
2. `.makefile`. Create a file called `.makefile` in your problem set directory with commands adjusted based on the template below. Then, you can run `make all` on your command line to build the files, and `./mapfold_tests.byte` to run the tests. Before you submit to Gradescrope, make sure to run `make clean` in your directory to remove all the `byte` files that resulted from the build.

```
all: ps2 ps2_tests

ps2: mapfold.ml
    ocamlbuild -use-ocamlfind mapfold.byte

ps2_tests: mapfold_tests.ml
    ocamlbuild -use-ocamlfind mapfold_tests.byte

clean:
    rm -rf _build *.byte
```