Implementing Bignums in μSmalltalk

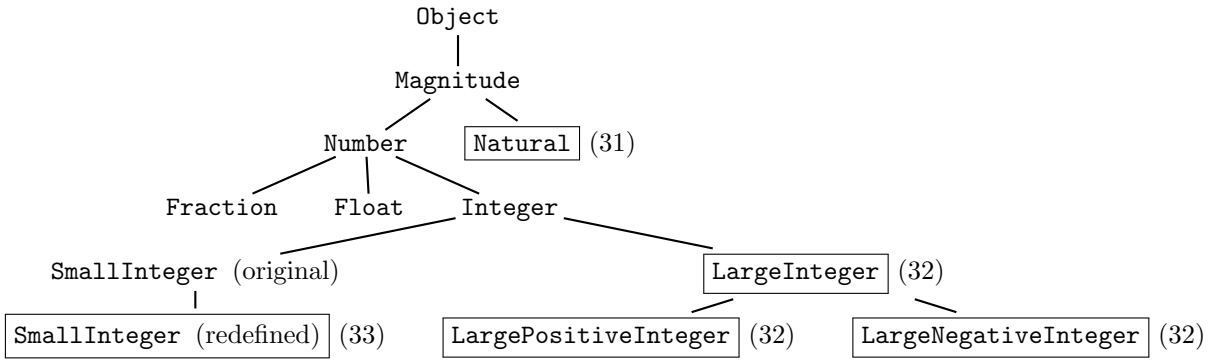
comp105-staff@cs.tufts.edu

Spring 2016

1 Approach

The pair portion of the μSmalltalk assignment is to implement arbitrary-precision arithmetic (Bignums). In completing this part of the assignment, particularly in writing your `Natural` class, you are faced with a great number of methods to implement. To help you organize this task, we suggest which methods to implement in what order, and we sketch how implementations of some methods may depend on other methods.

The diagram below shows what the class hierarchy will look like once you finish implementing bignums. The unboxed classes are from the initial basis of μSmalltalk. The boxed classes are new classes you will write for this assignment. Each class that is followed by a number is from the Exercise with that number.



2 Class `Natural`

We suggest you implement class `Natural` in three stages, testing extensively at the end of each stage.

Stage I — Basics

1. Methods `digit:`, `digit:put:`, and `makeEmpty:` manipulate the array of digits that represent the number. Implement these methods first.

   Remember that `digit:` should work with any nonnegative argument, no matter how large.

   Arrays in μSmalltalk are 1-based not 0-based: the first element is at position 1, not position 0. The mismatch between μSmalltalk's 1-based arrays and the 0-based abstraction you are using for polynomials is tedious. You should use the `digit:` and `digit:put:` methods to **hide** the 1-based nature of the underlying representation.

   If you use the `digit:` method carefully, you'll have to worry about sizes only when you allocate new results.

2. Method `doDigits:` has to do with *indexes*, not with digits themselves. This way it can be used for mutation, something you should test.

3. Once you have access to the digits, you can define `trim`, which removes unneeded leading zeroes.

4. Once `trim` is written, you can write `digits:`, to initialize a newly allocated bignum.

5. Now you can define the *class* method `new:`, which is your first public method—it creates a new bignum.

Stage II — Simple functions

6. Method `decimal` can be very simple if you use base $b = 10$. If you use a larger base, there will be some complexity here.

7. Once you have `decimal`, `print` is easy. Now you can debug!

8. Method `isZero` should be straightforward at this point.

9. With access to the digits, you can write `=`. You will find iteration over digits (`doDigits:`) to be helpful, and you will need to be careful when comparing bignums of different degree. (There is a very simple, elegant solution to the degree problem; try to find it!)

Stage III — Arithmetic

10. The heart of your arithmetic implementation will be the two methods `set:plus:` and `set:minus:`. They depend on the digit methods above. A loop driven by `doDigits:` may be helpful.

11. You can now implement addition with method `+`. In addition to `set:plus:`, you may find it useful to use `trim` and `makeEmpty:`.

12. Subtraction is more complicated because it can fail: the difference of two natural numbers is not always a natural number. But building `subtract:withDifference:ifNegative:` on top of `set:minus:` is otherwise analogous to your construction of `+`.

13. With natural-number subtraction in hand, you can now implement the standard methods `-` and `<`. You should be able to get everything you need from `subtract:withDifference:ifNegative:`, without having to use lower-level methods of class `Natural`.

14. Multiplication is the most complicated of all. You will want to allocate a new number with `makeEmpty:` and initialize it to zero. Then, as suggested in the book, you'll need a double sum to add in all the partial products. A doubly nested `doDigits:` loop will help. To manipulate the partial products, `digit:` and `digit:put:` will be essential. Finally, use `trim` to control the growth of your bignums.

3 `LargePositiveInteger` and `LargeNegativeInteger`

Once you've gotten this far, `LargePositiveInteger` and `LargeNegativeInteger` will be relatively straightforward. The list of methods and hints given in the book should get you through. You will lean heavily on your `Natural` methods, but only the *public* methods. These are the methods of class `Magnitude`, together with the methods listed in the box on page 980.

You will also want to reuse the `LargeInteger` methods as much as you can. The really interesting part of this problem is the technique of *double dispatch*. This technique will require that you add such methods as `addLargePositiveIntegerTo:`. You'll then use these to rewrite the basic arithmetic methods on integers.