COMP 105 Assignment: Smalltalk

Due Tuesday, April 26, at 11:59 PM.

The purpose of this assignment is to help you get acquainted with pure object-oriented programming. The assignment is divided into two parts.

- In the first part, you do a few small warmup problems to get you used to pure object-oriented style and to acquaint you with uSmalltalk's large initial basis.
- In the second part, you implement *bignums* in uSmalltalk. Bignums are useful in their own right, and they illustrate the important object-oriented technique of *double dispatch*.

You will find a uSmalltalk interpreter in `/comp/105/bin/usmalltalk`. This interpreter treats the variable `&trace` specially; by defining it with `val`, you can trace message sends and answers. It is an invaluable aid to debugging.

You will find useful sources are in the git repository, which you can clone by

```
  git clone linux.cs.tufts.edu:/comp/105/book-code
```

The repository `examples` directory includes copies of the initial basis, collection classes, financial history, and other examples from the textbook.

Helpful hint

Don't overlook the `protocol` and `localProtocol` methods which are defined on every *class*, as shown in the figure at the top of page 370.

Part I: Object-oriented warmup (to do on your own)

In Ramsey and Kamin, Chapter 9, do Exercises 4, 7(a), 14, and 27.
Difficulty: **

To solve all four problems, you shouldn't need to add or change more than 20 lines of code in total.

Part II: Bignums (to do with a partner)

In Ramsey and Kamin, Chapter 9, do Exercises 9, 10, 11 and Exercise **T** below. To simplify your life, you need not implement long division, and we recommend you choose base *b* = 10. There's significant extra credit available for experimenting with other bases.

My `Natural` class is over 100 lines of uSmalltalk code; my large-integer classes are 22 lines apiece. My modifications to predefined number classes are about 25 lines.
Difficulty: ****

Background

Sometimes you want to do computations that require more precision than you have available in a machine word. Full Scheme, Smalltalk, and Icon all provide ``bignums.'' These are integer implementations that automatically expand to as much precision as you need. Because of their dynamic-typing discipline, these languages make the transition transparently—you can't easily tell when you're using native machine integers and when you're using bignums. In uSmalltalk, the data abstraction can *almost* completely hide whether you have regular or extra-precision integers.

You will find bignums and the bignum algorithms discussed at some length in Dave Hanson's book and in the article by Per Brinch Hansen. Be aware that your assignment below differs significantly from the implementation in Hanson's book.

Notes and hints

- This is a big, complicated set of problems with a lot of methods. There is a handout online with suggestions about which methods depend on which other methods and in what order to tackle them.
- If you should choose to do the extra credit with large bases (below), remember that the private `decimal` method must return a list of **decimal** digits, even if base 10 is not what is used in the representation. Suppress leading zeroes unless the value of `Natural` is itself zero.
- You can think about borrowing code from Hanson's implementation (see also his distribution), but unless you've looked at the book you may be a bit overwhelmed. `XP_add` does add with carry. `XP_sub` does subtract with borrow. `XP_mul` does `z := z + x * y`, which is useful, but is not what we want unless `z` is zero initially. Moreover, Hanson has to pass all the lengths explicitly.
- Mutation is used heavily in Hanson's implementation, but the class `Natural` is an immutable type. Your methods must *not* mutate existing natural numbers; you can mutate only a newly allocated number that you are sure has not been seen by any client.
- If you use the `digit:` method carefully, you'll have to worry about sizes only when you allocate new results.
- Because classes are objects like any others, you can change most classes by *redefining* them, as the code in Ramsey and Kamin, chunk 455a redefines class `SmallInteger`. In order to make your solution work with an unmodified `usmalltalk`, **you must use this technique**.

Testing bignum arithmetic

To help you test your work, here is code that computes and prints factorials:

<fact.smt>=
```
(define factorial (n)
  (if (strictlyPositive n)
     [(* n (value factorial (- n 1)))]
     [1]))

(class Factorial Object
  ()
  (classMethod printUpto: (limit) (locals n nfac)
     (begin
        (set n 1)
        (set nfac 1)
        (while [(<= n limit)]
           [(print n) (print #!) (print space) (print #=) (print space) (println nfac)
            (set n (+ n 1))
            (set nfac (* n nfac))]))))
```

You might find it useful to test your implementation with the following table of factorials:

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

```
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
```

Be warned that **this test by itself is inadequate**. You will want other tests.

If you want to make comparisons with a working implementation of bignums, the languages Scheme, Icon, and Haskell all provide such implementations. (Be aware that the real Scheme `define` syntax is slightly different from what we use in uScheme.)

Exercise T

Submit one other test of your bignums in `bigtests.smt` and explain in what way the submitted test is superior to the factorial test. The question you should ask yourself is: *What will constitute an adequate test of bignums?*

Your `bigtests.smt` file should be formatted as follows:

1. It should begin with whatever definitions you need to run the test.
2. It should define a class `Test105` with a class method `run` that actually runs the test.
3. It should end with a **comment** that contains just the output from sending the `run` method to the `Test105` class, and nothing else

You `bigtests.smt` file should *not* include any code that *implements* bignums. Here is a trivial example:

<example bigtests.smt>=
```
(class Test105 Object
  ()
  (class-method run ()
     (locals n 10-to-the-n)
     (set n 0)
     (set 10-to-the-n 1)
     (whileTrue: [(< n 10)]
         [(set n (+ n 1))
          (set 10-to-the-n (* 10 10-to-the-n))])
     10-to-the-n)
)
; 10000000000
```

If this test is run in an unmodified interpreter, it breaks with an arithmetic overflow and a stack trace.

Extra-credit problem: Base variations

A key problem in the representation of integers is the choice of the base *b*. Today's hardware supports `b = 2` and sometimes `b = 10`, but when we want bignums, the choice of `b` is hard to make in the general case:

- If `b` = 10, then converting to decimal representation is trivial, but storing bignums requires lots of memory.
- The larger `b` is, the less memory is required, and the more efficient everything is.
- If `b` is a power of 10, converting to decimal is relatively easy and is very efficient. Otherwise it requires (possibly long) division.
- If `(b-1) * (b-1)` fits in a machine word, than you can implement multiplication in high-level languages without difficulty. *(Serious implementations pick the largest `b` such that `a[i]` is guaranteed to fit in a machine word, e.g., `2^32` on modern machines. Unfortunately, to work with such large values of `b` requires special machine instructions to support ``add with carry'' and 64-bit multiply, so serious implementations have to be written in assembly language.)*
- If `b` is a power of 2, bit-shift can be very efficient, but conversion to decimal is expensive. Fast bit-shift can be important in cryptographic and communications applications.

If you want signed integers, there are more choices: signed-magnitude and `b`'s-complement. Knuth volume 2 is pretty informative about these topics.

For extra credit, try the following variations on your implementation of class `Natural`:

1. Implement the class using an internal base *b*=10. Measure the time needed to compute the first 50 factorials.
2. Make an argument for the largest possible base that is still a power of 10. Change your class to use that base internally. (If you are both careful and clever, you should be able to change only the class method `base` and not any other code.) Measure the time needed to compute the first 50 factorials. Note both your measurements and your argument in your README file.

Because Smalltalk hides the representation from clients, a well-behaved client won't be affected by a change of base. If we wanted, we could take more serious measurements and pick the most efficient representation.

More Extra-credit problems

Division. Implement long division for `Natural` and for large integers. If this changes your argument for the largest possible base, explain how.

Largest base. Change the base to the largest reasonable base, not necessarily a power of 10. You will have to re-implement `decimal` using long division. *Measure* the time needed to compute *and print* the first 50 factorials. Does the smaller number of digits recoup the higher cost of converting to decimal?

Comparisons. Make sure comparisons work, even with mixed kinds of integers. So for example, make sure comparisons such as `(< 5 (* 1000000 1000000))` produce sensible answers.

Space costs. Instrument your `Natural` class to keep track of the size of numbers, and measure the space cost of the different bases. Estimate the difference in garbage-collection overhead for computing with the different bases, given a fixed-size heap.

Pi (hard). Use a power series to compute the first 100 digits of pi (the ratio of a circle's circumference to its diameter). Be sure to cite your sources for the proper series approximation and its convergence properties. *Hint: I vaguely remember that there's a faster convergence for pi over 4. Check with a numerical analyst.*

What to submit

Solo work

- A README file
- A file `finhist.smt` showing your solution to Exercise 14.
- A file `basis.smt` showing whatever changes you had to make to the initial basis to do Exercises 4, 7(a) and 27. Please be sure to identify your solutions using conspicuous comments, e.g.,

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;  solution to Exercise 4
(class Array ...
)
```

Pair work

- A `README` file that should include some indication of how you tested your bignum code for Part II. Of course we also want the usual stuff about your collaborators and your time spent.
- A file `bignum.smt` showing your solutions to Exercises 9, 10, and 11. This file **must** work with an *unmodified* `usmalltalk` interpreter. Therefore, if for example you use results from problems 4, 7(a), 11, or any other problem (e.g., the class method `from:` on the `Array` class), you will need to duplicate those results in `bignum.smt` as well as in `basis.smt` above.

- A file `bigtests.smt` showing your solutions to Exercise T.

  Submit code using `submit105-small-solo` and `submit105-small-pair`.

  - initialXi: D1