COMP 105 Assignment: Smalltalk

Due Monday, April 22, 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 Figure 10.5 on page 413.

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

In Ramsey, Chapter 10, do Exercises 4, 9, 10, and 29a on pages 508 to 510 and 515.
Estimated 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, Chapter 10, do Exercises 31, 32, and 33 on pages 516–519 and Exercise **T** below.
To simplify your life, you need not implement long division, and we recommend you choose base *b* = 10. If you want to experiment with other bases, you get significant extra credit.

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.
Estimated 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 (which should be free online to Tufts students) 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.
- It is *very* annoying that you are stuck with a 1-based array type to implement a 0-based abstraction (polynomials) for class `Natural`. I recommend that you use the `digit:` and `digit:put:` methods to **hide** the 1-based nature of the underlying representation.
- 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, page 519 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. Here is some advice

- Try testing your class `Natural` by generating a long, random string of digits, then computing the corresponding number using a combination of addition and multiplication by 10.
- If you don't have multiplication working yet, you can use the following sequence to multiply by 10:

  ```
  (set p n)       ; p == n
  (set p (+ p p)) ; p == 2n
  (set p (+ p p)) ; p == 4n
  (set p (+ p n)) ; p == 5n
  (set p (+ p p)) ; p == 10n
  ```

  This idea will test only your addition; if you have bugs there, fix them before you go on.
- You can write, in µSmalltalk itself, a method that uses the techniques above to convert a sequenceable collection of decimal digits into a natural number.
- For easy comparison, you can even define a *subclass* of `Array` or `List` that prints just the digits, with no spaces.
- Once you are confident that addition works, you can test subtraction of natural numbers by generating a long random sequence, then subtracting the same sequence in which all digits except the most significant are replaced by zero.
- You can create more ambitious tests of subtraction by generating random natural numbers and using the algebraic law *(m + n) - m = n*. You can also check to see that unless *n* is zero, *m - (m + n)* causes a run-time error.
- It is harder to test multiplication, but you can at least use repeated addition to test multiplication by *small* values. The `timesRepeat:` method is defined on any integer.
- You can also easily test multiplication by large powers of 10.
- You can use similar techniques to test large integers.

If you want more effective tests of multiplication and so on, compare your results 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.) We recommend you use `ghci` on the command line; standard infix syntax works. If you want something more elaborate, use Standard ML of New Jersey (command `sml`), which has an `IntInf` module that implements bignums.

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 contain a **summary characterization** of the test in at most 60~characters, formatted on a line by itself as follows:

   ```
   ; Summary: .........
   ```

   (Your summary should be a simple English phrase that describes your test. Examples might be ``Ackermann's function of (1, 1)," ``sequence of powers of 2," or ``combinations of +, *, and - on random numbers.")
3. It should define a class `Test105` with a class method `run` that actually runs the test.

4. It should end with **comments** that contain just the output from sending the `run` method to the `Test105` class, and nothing else. If the output is a single line, write a one-line comment. If the output takes multiple lines, put each line of output in a comment on its own line.
5. The expression `(run Test105)` **must take less than 2 CPU seconds to evaluate**.

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

<example bigtests.smt>=
```
; Summary: 10 to the tenth power
(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., `2^64` on modern machines. Unfortunately, to work with such large values of `b` requires special machine instructions to support ``add with carry'' and 128-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.*

Avoid Common Mistakes

Here are some common mistakes to avoid:

- It's common to overlook class methods. They are a good place to put information that doesn't change over the life of your program.
- It's surprisingly common for students to submit code for the small problems without ever even having run the code or loaded it into an interpreter. If you run even one test case, you will be ahead of the game.
- It's too common to submit bignum code without having tested all combinations of methods and arguments. Your best plan is to write a simple shell script that has three nested loops (over operator and both operands) and can emit test code either for your own interpreter or for an interpreter that has bignums built in, like `ghci`.
- It's a common mistake to submit a file `bigtests.smt` that has junk at the end or is otherwise badly formatted and thereby earns little credit.
- It's relatively common for students' code to make a false distinction between two flavors of zero. In integer arithmetic, there is only one zero, and it always prints as ``0''.

What to submit

Solo work

- A README file
- A file `finhist.smt` showing your solution to Exercise 9.
- A file `basis.smt` showing whatever changes you had to make to the initial basis to do Exercises 4, 29a, and 10. Please 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 31, 32, and 33. This file **must** work with an *unmodified* `usmalltalk` interpreter. Therefore, if for example you use results from Exercises 4, 29a, 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`.

How your work will be evaluated

All our usual expections for **form**, **naming**, and **documentation** apply. But in this assignment we will focus on **structure** and **clarity**.

	Exemplary	**Satisfactory**	**Must improve**
Structure	• An object's behavior is controlled by dispatching (or double dispatching) to an appropriate method of its class. • Code uses method dispatch instead of conditionals. • Mixed operations on different classes of numbers are implemented using double dispatch. • *Or*, mixed operations on different classes of numbers are implemented by arranging for the classes to share a common protocol. • *Or*, mixed operations on different classes of numbers are implemented by arranging for unconditional coercions. • Code deals with exceptional or unusual conditions by passing a suitable `exnBlock` or other block. • Code achieves new functionality by reusing existing methods, e.g., by sending messages to `super`. • *Or*, code achieves new functionality by adding new methods to old classes to respond to an existing protocol. • Code that works with collections works with *any* `Collection` class.	• An object's behavior is influenced by interrogating it to learn something about its class. • Code contains one avoidable conditional. • Mixed operations on different classes of integers involve explicit conditionals. • Code protects itself against exceptional or unusual conditions by using Booleans. • Code contains methods that appear to have been copied and modified. • The base used for natural numbers appears in exactly one place, but code that depends on it knows what it is, and that code will break if the base is changed in any way. • Overflow is detected only by assuming the number of bits used to represent a machine integer, but the number of bits is explicit in the code.	• Code contains case analysis or a conditional that depends on the class of an object. • Code contains more than one avoidable conditional. • Mixed operations on different classes of integers are implemented by interrogating objects about their classes. • Code copies methods instead of arranging to invoke the originals. • Code that is supposed to works with all collections works only with some subclasses of collections. • The base used for natural numbers appears in multiple places. • Overflow is detected only by assuming the number of bits used to represent a machine integer, and the number of bits is *implicit* in the value of some frightening decimal literal.

	• The base used for natural numbers appears in exactly one place, and all code that depends on it consults that place. • *Or*, the base used for natural numbers appears in exactly one place, and code that depends on either consults that place or assumes that the base is some power of 10 • No matter how many bits are used to represent a machine integer, overflow is detected by using appropriate primitive methods, not by comparing against particular integers.		
Clarity	• Course staff see no more code than is needed to solve the problem. • Course staff see how the structure of the code follows from the structure of the problem.	• Course staff see somewhat more code than is needed to solve the problem. • Course staff can relate the structure of the code to the structure of the problem, but there are parts they don't understand.	• Course staff roughly twice as much code as is needed to solve the problem. • Course staff cannot follow the code and relate its structure to the structure of the problem.
Documentation	• Private methods are documented with contracts. • *Or*, private methods use exactly the contracts suggested in the bignums handout.		• Private methods are neither documented nor consistent with the bignums handout.