Object-Oriented Programming in Smalltalk

COMP 105 Assignment

Due Tuesday, May 1, 2018 at 11:59PM

Contents

Overview

Setup

Interactive debugging with μSmalltalk

Reading comprehension (10 percent)

Individual Problem

Pair Problems: Bignum arithmetic

Big picture of the solution

Unit testing bignums in Smalltalk

How to prepare your code for our testing

Details of all the problems

A simple sanity check

More advice about testing natural numbers

Other hints and guidelines

Avoid common mistakes

Extra credit

What and how to submit: Individual problem

What and how to submit: Pair problems

How your work will be evaluated
Overview

The purpose of this assignment is to familiarize you with pure object-oriented programming, in which even simple algorithms send lots of messages back and forth among a cluster of cooperating, communicating objects. The assignment is divided into three parts.

- You begin with reading comprehension.
- You do a small warmup problem, which acquaints you with pure object-oriented style and with \( \mu \text{Smalltalk} \)'s large initial basis.
- You implement \textit{bignums} in \( \mu \text{Smalltalk} \). As in the SML assignment, you will implement both natural numbers and signed integers. You will also use object-oriented dispatch to implement “mixed arithmetic” of large and small integers—a useful abstraction that demonstrates the “open” nature of true object-oriented systems.

This assignment is time-consuming. Many students have experience in languages called “object-oriented,” but few students have experience with the extensive, pervasive inheritance that characterizes idiomatic Smalltalk programs.

Setup

The \( \mu \text{Smalltalk} \) interpreter is in \texttt{/comp/105/bin/usmalltalk}. Many useful \( \mu \text{Smalltalk} \) sources are included the book’s git repository, which you can clone by

\begin{verbatim}
git clone homework.cs.tufts.edu:/comp/105/build-prove-compare
\end{verbatim}

Sources that can be found in the \texttt{examples} directory includes copies of predefined classes, collection classes, financial history, and other examples from the textbook.

Interactive debugging with \( \mu \text{Smalltalk} \)

Smalltalk is a little language with a big initial basis; there are lots of predefined classes and methods. To help you work with the big basis, as well as to debug your own code, we recommend two tools:

- Every \texttt{class} understands the messages \texttt{protocol} and \texttt{localProtocol}. These methods are shown in Figure 10.8 on page 694. They provide a quick way to remind yourself what messages an object understands and how the message names are spelled.
- The interpreter can help you debug by emitting a \texttt{trace} of up to \( n \) message sends and answers. Just enter the definition

\begin{verbatim}
(val &trace n)
\end{verbatim}

at the read-eval-print loop. To turn tracing off, \( (\text{set } &\text{trace } 0) \).

Here is an (abbreviated) example trace of message \texttt{new} sent to class \texttt{List}, which is the subject of one of the reading-comprehension questions:

\begin{verbatim}
-> (new List)
standard input, line 3: Sending message (new) to class List
\end{verbatim}
predefined, line 478: Sending message (new) to class SequenceableCollection
predefined, line 478: (new <class List>) = <List>
predefined, line 478: Sending message (new) to class ListSentinel
predefined, line 469: Sending message (new) to class Cons
predefined, line 469: (new <class ListSentinel>) = <ListSentinel>
predefined, line 470: Sending message (pred: <ListSentinel>) to an object of class ListSentinel
predefined, line 470: (pred: <ListSentinel> <ListSentinel>) = <ListSentinel>
predefined, line 471: Sending message (cdr: <ListSentinel>) to an object of class ListSentinel
predefined, line 471: (cdr: <ListSentinel> <ListSentinel>) = <ListSentinel>
predefined, line 478: (new <class ListSentinel>) = <ListSentinel>
predefined, line 478: Sending message (sentinel: <ListSentinel>) to an object of class List
predefined, line 478: (sentinel: <List> <ListSentinel>) = <List>
standard input, line 3: (new <class List>) = <List>
<trace ends>
List( )

One warning: as usual, the interpreter responds to an expression evaluation by printing the result. But in Smalltalk, printing is achieved by sending the println to the result object. This interaction is also traced, and the results can be startling. Here, for example, is the result of evaluating the literal integer 3:

-> 3

internally generated SEND node, line 1: Sending message (println) to object of class SmallInteger
internal expression "((begin (print self) (print newline) self))", line 1: Sending message (print) to
3 internal expression "((begin (print self) (print newline) self))", line 1: (print 3<SmallInteger>) =
internal expression "((begin (print self) (print newline) self))", line 1: Sending message (print) to
predefined classes, line 229: Sending message (printu) to object of class SmallInteger

predefined classes, line 229: (printu 10<SmallInteger>) = 10<SmallInteger>
internal expression "((begin (print self) (print newline) self))", line 1: (print <Char>) = 10<SmallInteger>
internally generated SEND node, line 1: (println 3<SmallInteger>) = 3<SmallInteger>

As you see, even a simple operation like printing a number involves four message sends. Don’t let them confuse you.

**Reading comprehension (10 percent)**

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions.

1. **Receivers, arguments, and messages.** Read the first seven pages of chapter 10, through section 10.1.3. Now examine these expressions from the definition of class Tikzpicture, which should be below Figure 10.3 on page 676:

   \begin{verbatim}
   (div: w 2)
   (drawOn: shape self)
   (do: shapes [block (shape) (drawOn: shape self)])
   \end{verbatim}

1. /cqs.small.txt
In each expression, please identify the receiver, the argument, and the message:

In (div: w 2),
- The receiver is …
- The argument is …
- The message is …

In (drawOn: shape self),
- The receiver is …
- The argument is …
- The message is …

In (do: shapes [block (shape) (drawOn: shape self)]),
- The receiver is …
- The argument is …
- The message is …

2. Colons in method names. Continuing with the analysis of Tikzpicture, in both the protocol and the implementation, method `add:` has one colon in the name, method `draw` has no colons in the name, and the method `drawEllipseAt:width:height:` has three colons in the name.

- What, if anything, does the number of colons have to do with `receivers`?
  
  Your answer: …

- What, if anything, does the number of colons have to do with `arguments`?
  
  Your answer: …

If you need to, review the presentation in section 10.1.1 on “Objects and Messages,” which shows messages sent to shapes.

3. Class protocols and instance protocols. Every message is part of some protocol. As example messages, study the transcript in code chunks 671e and 672, which puts three shapes into a picture and then draws the picture.

   (a) Of the messages used in the transcript, which ones are part of the class protocol for Tikzpicture, and which are part of the instance protocol?

   (b) In general, what do you do with messages in a class protocol, and how does that differ from what you do with messages in an instance protocol?

4. Dynamic dispatch, part I: a toy class. For the mechanisms of message send and dynamic dispatch, read section 10.3.4, which starts on page 689. Using the class definitions in that section, message `m1` is sent to an object of class `C`. What method `definitions` are dispatched to, in what order?

Please edit this answer to put in the correct methods and classes:

- Dispatch to method `m1` on class?
- Dispatch to method `?` on class? …

5. Dynamic dispatch, part II: number classes. Study the implementation of class `Number`, which starts around page 745. Now study the implementation of class `Fraction`, which starts around page 749.
When message - (minus) is sent to the Fraction (/ 1 2) with argument Fraction (/ 1 3), the computation dispatches message to instance methods of classes Fraction, Number, and Small-Integer, as well as a class method of class Fraction. We are interested in only some of those dispatches—ones that meet both of these criteria:

- The message is sent from a method defined on class Fraction or class Number.
- The message is received by an instance of class Fraction or class Number.

These criteria rule out class methods of class Fraction, messages sent to SmallInteger, and so on.

Starting with message - (minus) is sent to an instance of Fraction, please identify only the interesting dispatches:

<table>
<thead>
<tr>
<th>Message</th>
<th>Sent from method defined on class</th>
<th>Sent to object defined on class</th>
<th>Method defined on class</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (anywhere)</td>
<td>Fraction</td>
<td>Number</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>Number</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

... complete the rest of this table ...

6. Dynamic dispatch, part III: messages to self and super. Now study the class method new defined on class List, which appears just after page 739. The definition sends message new to super. (Keep in mind: because new is a class method, both super and self stand for the class, not for any instance.)

(a) When class method new is executed, what three messages are sent by the method body, in what order? (If you like, you can also study the message trace shown above, but it may be simpler just to look at the source code.)

(b) What does each of the three messages sends accomplish?

(c) If we change new’s definition so instead of (new super) it says (new self), which of the following scenarios best describes how the changed program behaves?

1) The new message will be dispatched to class List. The same method will run again, and the computation will not terminate.

2) The new message will be dispatched to a different class, and the reply to the new message will leave the sentinel pointing to the wrong value.

3) Nothing will change; in this example, there’s no difference between (new super) and (new self).

Your answer: The best description is scenario number ?

7. Design of the numeric classes. Read about coercion in section 10.4.6 on page 707. Look at the last part of the instance protocol for Number on page 706. Explain the roles of the methods asInteger, asFraction, asFloat, and coerce: If you are unsure, look at the implementations of these methods on class Integer, starting on page 747.

The role of asInteger is ...

The role of asFraction is ...
The role of `asFloat` is ...  
The role of `coerce:` is ...  

You are ready to implement mixed arithmetic, with coercions, in exercise 44.

8. **Abstract classes in principle.** In section 10.10.1, which starts on page 792 (“Key words and phrases”), you will find a short definition of “abstract class.” What is the *purpose* of an abstract class? Pick one of the responses below.

   (a) To hide the representation of instances so programmers can change internal details without affecting client code  
   (b) To define methods that other classes inherit, so that subclasses get useful default methods  
   (c) The same as the purpose of a regular class: to define an abstraction

Your answer: …

9. **Abstract classes in practice: magnitudes and numbers.** Your natural-number class will inherit from abstract class `Magnitude`, and your big-integer code will inherit from `Magnitude` and from `Number`, which is also an abstract class.

   (a) Study the implementation of class `Magnitude`; it is the first dozen lines of code in section 10.6.6, which starts on page 745. List all the methods that are “subclass responsibility”:

   Your answer: …

   These are methods that you must implement in both your `Natural` class and your large-integer classes.

   (b) The very next class definition is the definition of abstract class `Number`. Read the first code chunk and again, list all the methods that are “subclass responsibility”:

   Your answer: …

   These are the methods that you must implement in your large-integer classes. (Two of them, `+` and `*`, must also be implemented in class `Natural`.)

You are getting ready to implement large integers.

10. **Double Dispatch.** Read section 10.6.5, which starts on page 744. Of the methods listed in the previous question, list each one that needs to know *either* of the following facts about its *argument* (not its receiver):

    • Whether the argument is large or small  
    • If the argument is large, whether it is “positive” or “negative”

For example, `+` is such a method.

   (a) Please list all such methods here:

   Your answer: `+` …

   (b) The methods in part (a) are exactly the ones that require double dispatch. The implementation of each such method sends a message to its *argument*, and the exact message depends on the class of the *receiver*. 
Assume that the receiver is a LargePositiveInteger. Please say, for each method in part (a), what message the method implementation sends to the argument.

Your answer:

Method + sends addLargePositiveIntegerTo: to the argument

You are ready to implement large integers (exercise 43).

Individual Problem

Working on your own, please solve exercise 39(a) on page 810 of Build, Prove, and Compare. This exercise is a warmup designed to prepare you for the Bignum problems in the pair portion of the assignment.


When the problem says “Arrange the Fraction and Integer classes”, the text means to revise one or both of these classes or define a related class. If you revise an existing class, you must do so without changing the source code. For an example, if you want to revise class SmallInteger, you must redefine class SmallInteger using the idiom on page 814:

(class SmallInteger SmallInteger
()
... new or revised method definitions ...
)

This idiom enables you to change predefined classes without editing the source code of the \( \mu \)Smalltalk interpreter. Using the idiom as needed, you should be able to put your entire solution in file frac-and-int.smt.

Hints:

- At minimum, your solution should support addition, subtraction, and multiplication, so include at least one check-expect unit test for each of these operations. These tests are run only on your own code, so they do not have to be formatted in any special way.

- In a system with abstract data types, you can’t easily mix integers and fractions; they have different types. But in an object-oriented system with behavioral subtyping, you just have to get one object to “behave like” another—which means implementing its protocol. In some cases, this might include implementing private methods.

- If you change class Integer, this change doesn’t affect class SmallInteger, which continues to inherit from the original version of Integer. So if you change Integer, count on changing SmallInteger as well.

Related reading:

- For an overview of the Magnitude class and its relationship to numbers, read the first page of section 10.4.6, which starts on page 707. Also in that section, read about Integer and Fraction.
• For the implementation of Integer, see page 747. The implementation of class SmallInteger is also nearby, but for the time being, you can ignore the details of how it is implemented—almost all the methods are primitive.

• For the implementation of Fraction, see page 749. Study the implementation of method +, and observe how it relies on the exposure of representation through private methods num and den.

• If nothing comes to you, try reading about how we get access to multiple representations in the object-oriented way: section 10.6.5, which starts on page 744. You will need to read this section later anyway.

How big is it? You shouldn’t need to add or change more than 10 lines of code in total. The optimal solution is no more than a few lines long.

Pair Problems: Bignum arithmetic

For these problems, you may work with a partner. Please solve exercise 42 on page 811, exercise 43 on page 813, and exercise 44 on page 814 of Build, Prove, and Compare, and exercise T below. You do not need to implement long division.

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,” which automatically expand to as much precision as you need. Unlike languages that use abstract data type, Scheme, Smalltalk, and Icon make the transition from machine integers to bignums transparent—from the source code, it’s not obvious when you’re using native machine integers and when you’re using bignums. You will build transparent bignums in Smalltalk.

Big picture of the solution

Smalltalk code sends lots of messages back and forth among lots of methods that are defined on different classes. This model shows both the power of Smalltalk—you get a lot of leverage and code reuse—and the weakness of Smalltalk—every algorithm is smeared out over half a dozen methods defined on different classes, making it hard to debug. But this is the object-oriented programming model that lends power not just to Smalltalk but also to Ruby, Objective-C, Swift, Self, JavaScript, and to a lesser extent, Java and C++. To work effectively in any of these languages, one needs the big picture of which class is doing what. A good starting point is the Smalltalk bignums handout.²

Unit testing bignums in Smalltalk

Arithmetic comparisons should be tested using check-assert in the usual way. But other arithmetic operations can’t easily be tested using check-expect, because the parser handles only machine integers. To test these operations, especially on natural numbers, we recommend you use check-expect with the decimal method:

²../handouts/bignums.pdf
(check-expect (decimal (x:to: Power 10 60))
  '( 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(check-expect (decimal (x:to: Power 2 64))
  '( 1 8 4 4 6 7 4 4 0 7 3 7 0 9 5 5 1 6 1 6 ))
(check-expect (decimal (negated (x:to: Power 2 64)))
  '( - 1 8 4 4 6 7 4 4 0 7 3 7 0 9 5 5 1 6 1 6 ))

You may notice something a little strange here: the decimal method answers a List object, but these tests use Array objects. How can the tests pass? The tests pass because check-expect compares objects using similarity, not equality. The difference is explained in detail in section 10.3.5, which starts on page 689. The details probably are not necessary for this assignment, but when you are debugging, you may need to know that check-expect uses the similar: method, not the = method.

How to prepare your code for our testing

Many of our tests interact directly with the μSmalltalk interpreter. Our testing infrastructure enters definitions and looks at the responses. To pass our tests, you must define print methods that render numbers as normal people expect to see them. You cannot simply send decimal to self and print the result—you must print the individual digits, possibly preceded by a minus sign. For a Natural number, the print operation could be as simple as

(do: (decimal self) (block (digit) (print digit)))

Details of all the problems

42. Implementing arbitrary-precision natural numbers. Do exercise 42 on page 811 of Build, Prove, and Compare. Modify the protocol given in the assignment so that the div: method implements only short division: it should divide a Natural by a small integer, not by another Natural.

The choice of a base for natural numbers is yours. But for full credit, you must choose a base larger than 10.

The algorithms are the same algorithms you would have used on the sm\l assignment. And although the book assumes you will represent a natural number using an array, you are free to use a list of digits instead. Just be aware that unlike Standard ML, μSmalltalk has no immutable list abstraction. If you want a list of digits, follow the recommendations in the bignums handout, which explains how to roll your own.

Please do adapt your code from the sm\l assignment. Or if you prefer, you may adapt my solutions. (I provide both an array-based solution and a list-based solution.) Whatever code you adapt, be sure to attribute the source!

How big is it? I’ve written two implementations of class Natural:

---

3\sml.html
4\../handouts/bignums.pdf
5\../solutions/anatural.pdf
6\../solutions/fnatural.pdf
• Using the array representation and the hints in the book, my solution is about 120 lines of $\mu$Smalltalk code.

• Using the list-based representation and the hints in the bignums handout, my solution is about 150 lines of $\mu$Smalltalk code.

**Should I use a list or an array?** From the ML modules solutions, you can compare what list-based and array-based solutions look like. But those comparisons may be misleading: while the array-based solutions are roughly similar, the different way that case analysis is expressed using objects (versus pattern matching) makes the list-based solutions more different. For Smalltalk, in my opinion, neither of these solutions is strictly better than the other:

• The list-based solution is easier to get right but harder to get started. The easy part is that, as usual with lists, there are only two cases to deal with: an empty list and a nonempty list. The hard part is figuring out how to do the case analysis entirely with object-oriented dispatch—and how to avoid double dispatch.

• The array-based solution is harder to get right but easier to get started. The easy part is that there is no case analysis in the data: every natural number is represented by an array. The hard part is figuring out array lengths, managing array indices, and making sure not to confuse the index of a digit with the digit itself.

On the whole, I slightly prefer my list-based solution: although it is a little longer, I can look at any part and be confident that I understand exactly what it is doing. My array-based solution does not give me quite that same level of confidence.

**Related reading:**

• There is a detailed implementation guide in the bignums handout. It covers issues both array and list representations.

• In a system with abstract data types, binary operations like + and * are easy: you automatically have access to both representations. In a system with objects, not so! To learn how to get access to multiple representations in the object-oriented way, read section 10.6.5, which starts on page 744.

• Class Natural is a subclass of Magnitude. Study the Magnitude protocol in section 10.4.6. For information about the implementation of Magnitude, which should provide useful ideas about Natural, as well as the “subclass responsibilities,” study the implementation of Magnitude on page 745.

• For the interface to a Smalltalk array, study the Collection protocol in section 10.4.5, which starts on page 697. You have access to the protocol in Figure 10.10 on page 700, but you are more likely to use the KeyedCollection protocol in Figure 10.11 on page 702, especially at: and at:put:. Don’t overlook the Arrays section on pages 704 and 705, including its description of the Array class methods new: and from:.

• For list construction, which you will need for the decimal method, look at the List protocol in section 10.4.5, especially Figure 10.13 on page 704.

---

7 ../handouts/bignums.pdf
8 ../solutions/index.html#sml
9 ../handouts/bignums.pdf
10 Note: an object of class Natural is not a Number as Smalltalk understands it. In particular, class Natural does not support methods negated or reciprocal.
43. Implementing arbitrary precision integers. Do exercise 43 on page 813 of Build, Prove, and Compare. You need not implement the div: method. But in order to facilitate testing, you must implement a decimal method like the one you implemented on class Natural. An implementation is shown in the bignums handout11.

Because you build large integers on top of Natural, you don’t have to think about array or list representations any more. Instead you must focus on dynamic dispatch and on getting information from where it is to where it is needed.

The book has starter code for class LargeInteger, which you can copy (with acknowledgement) from /comp/105/build-prove-compare/examples/usmalltalk/large-int.smt.

How big is it? My solutions for the large integer classes are 22 lines apiece.

Related reading: This problem is all about dynamic dispatch, including double dispatch. Read section 10.6.5, which starts on page 744. (You’ll also have a chance to practice double dispatch in recitation.)

44. Modifying SmallInteger so operations that overflow roll over to infinite precision. Do exercise 44 on page 814 of Build, Prove, and Compare.

You must modify SmallInteger without editing the source code of the µSmalltalk interpreter. To do so, you will redefine class SmallInteger using the idiom on page 814:

```lisp
(class SmallInteger SmallInteger ()
  ... new method definitions ...
)
```

This idiom modifies the existing class SmallInteger; it can both change existing methods and define new methods. This code changes the basic arithmetic operations that the system uses internally. If you have bugs in your code, the system will behave erratically. At this point, you must restart your interpreter and fix your bugs. Then use the idiom again.

How big is it? My modifications to the SmallInteger class are about 25 lines.

Related reading: Everything about dispatch and double dispatch still applies. In addition, you need to know how overflow is handled using “exception blocks.”

- Review the presentation of blocks, especially the parameterless blocks (written with curly braces) in section 10.4.3, which starts on page 695.

- Read the description of at:ifAbsent: in the keyed-collection protocol in Figure 10.11 on page 702. Now study this expression:

```lisp
  (at:ifAbsent: '(0 1 2) 99 {0})
```

This code attempts to access element 99 of the array ( 0 1 2 ), which is out of bounds because the array only has only 3 elements. When given an index out of bounds, at:ifAbsent: sends value to the “exception block” {0}, which ultimately answers zero.

- Study the implementation of the at: method in ⟨other methods of class KeyedCollection 735d⟩, which uses at:ifAbsent: with an “exception block” that causes a run-time error if value is sent to it.

---

11 ../handouts/bignums.pdf
• Finally, study the overflow-detecting primitive methods in exercise 44 on page 814, and study the implementation of addSmallIntegerTo: in the code chunk immediately below. That is the technique you must emulate.

T. Testing Bignums. In standalone file bigtests.smt, you will write 9 tests for bignums:

• 3 tests will test only class Natural.
• 3 tests will test the large-integer classes, which are built on top of class Natural.
• 3 tests will test mixed arithmetic and comparison involving both small and large integers.

These tests will be run on other people's code, and they need to be structured and formatted as follows:

1. The test must begin with a summary characterization of the test in at most 60 characters, formatted on a line by itself as follows:
   ; Summary: ........

   The summary must be a simple English phrase that describes the test. Examples might be “Ackermann’s function of (1, 1),” “sequence of powers of 2,” or “combinations of +, *, and - on random numbers.”

2. Code must compute a result of class Natural, LargePositiveInteger, or LargeNegativeInteger. The code may appear in a method, a class method, a block, or wherever else you find convenient. The code must be included in file bigtests.smt.

3. The expected result must be written as a literal array, like one of the following:
   '1 2 3 4 9
   '-6 5 5 3 6

4. The test itself must be written using check-expect, sending the decimal message to the result computed in part 2, and expecting it to be similar to the literal array from part 3.

Each test must take less than 2 CPU seconds to evaluate.

Here is a complete example containing two tests:

; Summary: 10 to the tenth power, linear time, mixed arithmetic
(class Test10Power Object
  ()
  (class-method run: (power)
    [locals n 10-to-the-n]
    (set n 0)
    (set 10-to-the-n 1)
    (whileTrue: {(< n power})
      {set n (+ n 1))
        (set 10-to-the-n (* 10 10-to-the-n)))
      10-to-the-n)
  )
  (check-expect (decimal (run: Test10Power 10))
    '1 0 0 0 0 0 0 0 0 0)
  ; Summary: 10 to the 30th power, mixed arithmetic
  (check-expect (decimal (run: Test10Power 30))
    '1 0 0 0 0 0 0 0 0 0)
Here is another complete example:

; Summary: 20 factorial
(define factorial (n)
  (ifTrue: ifFalse: (strictlyPositive n)
    {{(* n (value factorial (- n 1)))}
     {1}})

(check-expect (decimal (value factorial 20))
  '( 2 4 3 2 9 0 2 0 0 8 1 7 6 6 4 0 0 0 ))

Related reading: No special reading is recommended for the testing problem. As long as you understand the examples above, that should be enough.

A simple sanity check

As a test, the factorial and power functions have grave limitations:

- These tests only ever multiply large numbers. They do not add, subtract, negate, or compare numbers.
- They never multiply two large numbers. They only ever multiply a large number by a small number, or two small numbers.

These properties of power and factorial make them poor tests of correctness, but they do make good initial sanity checks. Here, for example, is code that computes and prints factorials:

(class Factorial Object
  ()
  (class-method printUpto: (limit) [locals n nfac]
    (set n 1)
    (set nfac 1)
    (whileTrue: {(<= n limit)}
      {{(print n) (print '!') (printu 32) (print '=' (printu 32) (println nfac)
        (set n (+ n 1))
        (set nfac (* n nfac))})}})

As a sanity check sending (printUpto: Factorial 25) should print the following table of factorials:

<table>
<thead>
<tr>
<th>n</th>
<th>factorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
</tr>
</tbody>
</table>

\[pages/factorial.smt\]
More advice about testing natural numbers

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. You can generate a string of random digits on the command line by launching the bash shell and running this command:

```
for ((i = 0; i < 20; i++)); do echo -n ' ' $((RANDOM % 10)); done; echo
```

You can generate a test command from a list of digits using µScheme\(^{13}\):

```
(define nat-test (ns)
  (letrec ([exp-of (lambda (ns)
                     (if (null? ns)
                         0
                         (list3 '+ (car ns) (list3 '* 10 (exp-of (cdr ns))))))])
    (list4 'check-expect (list2 'decimal (exp-of (reverse ns)))
    ' '# ns)))) ;; for uSmalltalk, change output # to '
```

For example,

```
-> (nat-test '(1 2 3))
(check-expect (decimal (+ 3 (* 10 (+ 2 (* 10 (+ 1 (* 10 0))))))) # (1 2 3))
```

You will need to change the # mark to a quote mark, and you will need either to add method decimal to SmallInteger or to use a larger test case.

If you don’t have multiplication working yet, you can use the following class\(^{14}\) to multiply by 10:

```
(class Times10 Object
  ()
  (class-method by: (n) (locals p)
```
This idea will test only your addition; if you have bugs there, fix them before you go on.

You can write, in μSmalltalk instead of μScheme, a method that uses the techniques above to convert a sequenceable collection of decimal digits into a natural number.

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 on class `Natural`.

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.

Other hints and guidelines

**Start early.** Seamless arithmetic requires in-depth cooperation among about eight different classes (those you write, plus `Magnitude`, `Number`, `Integer`, and `SmallInteger`). This kind of cooperation requires aggressive message passing and inheritance, which you are just learning. There is a handout online\(^{15}\) with suggestions about which methods depend on which other methods and in what order to tackle them.

The bignums algorithms are the same as in the ML modules assignment, and in addition to consulting those solutions, you can consult the same references that are recommended in that assignment. In particular, Dave Hanson’s book\(^{16}\) discusses bignums and bignum algorithms at some length. It should be free online to Tufts students. You can think about borrowing code from Hanson’s implementation\(^{17}\) (see also his distribution\(^{18}\)). Be aware though that your assignment differs significantly from his code and unless you have read the relevant portions of the book, you may find the code overwhelming.

\(^{15}\)../handouts/bignums.pdf

\(^{16}\)../readings/indexbody.html#cii

\(^{17}\)xp.c

\(^{18}\)http://www.cs.princeton.edu/software/cii
• In Hanson’s code, \( \text{XP}_\text{add} \) does add with carry. \( \text{XP}_\text{sub} \) does subtract with borrow. \( \text{XP}_\text{mul} \) does \( z := z + x \times y \), which is useful, but is not what we want unless \( z \) is zero initially.

• Hanson passes all the lengths explicitly, which would not be idiomatic in \( \mu \text{Smalltalk} \).

• Hanson’s implementation uses mutation extensively, but the class \text{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 do emulate Hanson’s code, acknowledge him in your README file.

**Avoid common mistakes**

Below you will find some common mistakes to avoid.

It is 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 not common, but it’s a terrible mistake to try to implement mixed operations by interrogating an object about its class—a so-called “run-time type test.” Run-time type tests destroy behavioral subtyping. If you find yourself wanting to ask an object what its class is, seek help immediately.

It is surprisingly common for students to submit code for small problems without 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 is too common to submit bignum code without having tested all combinations of methods and arguments. Your best plan is to write a program, in the language if your choice, that loops over operator and both operands and generates at least one test case for every combination. Because \( \mu \text{Smalltalk} \) is written using S-expressions, you could consider writing this program in \( \mu \text{Scheme} \)—but any language will do.

It is 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”.

It’s surprisingly common to fail to tag the test summary with the prefix Summary:, or to forget it altogether.

**Extra credit**

Seamless bignum arithmetic is an accomplishment. But it’s a long way from industrial. The extra-credit problems explore some ideas you would deploy if you wanted everything on a more solid foundation.

**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 \( (b-1) \times (b-1) \) fits in a machine word, than you can implement multiplication in high-level languages without difficulty. (Serious implementations pick the largest \( b \) such that one digit fits 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’s *Art of Computer Programming, 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. Determine the largest possible base that is still a power of 10. Explain your reasoning. 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 and print the first 50 factorials.

3. Determine the largest possible base that is a power of 2. Explain your reasoning. Change your class to use that base internally. Measure the time needed to compute and print the first 50 factorials. Does having fewer digits recoup the higher cost of converting to decimal?

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.

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.

Write up your arguments and your measurements in your README file.

**Long division.** Implement long division for `Natural` and for large integers. If this changes your argument for the largest possible base, explain how. This article\(^{20}\) by Per Brinch Hansen describes how to implement long division.

**Mixed 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 and how to submit: Individual problem

Submit these files:

\(^{19}\) ../readings/indexbody.html#knuth2
\(^{20}\) ../readings/indexbody.html#division
• A README file containing
  – The names of the people with whom you collaborated
• A file cqs.small.txt containing your answers to the reading-comprehension questions
• A file frac-and-int.smt showing whatever definitions you used to do exercise 39(a). It probably includes new definitions (or redefinitions) of one or more of these classes: Fraction, Integer, and SmallInteger. And it most definitely includes at least three unit tests.

Please identify your solutions using conspicuous comments, e.g.,

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;
;;;;; Solution to Exercise XXX
(class Array ...
)

As soon as you have the files listed above, run submit105-small-solo to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

What and how to submit: Pair problems

Submit these files:
• A README file containing
  – A description of how you tested your bignum code
  – The names of the people with whom you collaborated
  – The numbers of the problems that you solved (including any extra credit)
  – Narrative and measurements to accompany your extra-credit answers, if any
• A file bignum.smt showing your solutions to Exercises 42, 43, and 44. This file must work with an unmodified usmalltalk interpreter. Therefore if you use results from exercise 39(a), or any other problem, you will need to duplicate those modifications in bignum.smt.
• A file bigtests.smt containing your solution to Exercise T.

As soon as you have the files listed above, run submit105-small-pair to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

How your work will be evaluated

All our usual expectations for form, naming, and documentation apply. But in this assignment we will focus on clarity and structure. To start, we want to be able to understand your code.
<table>
<thead>
<tr>
<th>Clarity</th>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>● Course staff see no more code than is needed to solve the problem.</td>
<td>● Course staff see somewhat more code than is needed to solve the problem.</td>
<td>● Course staff see roughly twice as much code as is needed to solve the problem.</td>
</tr>
<tr>
<td></td>
<td>● Course staff see how the structure of the code follows from the structure of the problem.</td>
<td>● Course staff can relate the structure of the code to the structure of the problem, but there are parts they don’t understand.</td>
<td>● Course staff cannot follow the code and relate its structure to the structure of the problem.</td>
</tr>
</tbody>
</table>

Structurally, your code should hide information like the base of natural numbers, and it should use proper method dispatch, not bogus techniques like run-time type checking.
<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The base used for natural numbers appears in exactly one place, and all code that depends on it consults that place.</td>
<td>• 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.</td>
<td>• The base used for natural numbers appears in multiple places.</td>
</tr>
<tr>
<td>• 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</td>
<td>• 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.</td>
<td>• 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.</td>
</tr>
<tr>
<td>• 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.</td>
<td>• Code contains one avoidable conditional.</td>
<td>• Code contains more than one avoidable conditional.</td>
</tr>
<tr>
<td>• Code uses method dispatch instead of conditionals.</td>
<td>• Mixed operations on different classes of integers involve explicit conditionals.</td>
<td>• Mixed operations on different classes of integers are implemented by interrogating objects about their classes.</td>
</tr>
<tr>
<td>• Mixed operations on different classes of numbers are implemented using double dispatch.</td>
<td>• Code protects itself against exceptional or unusual conditions by using Booleans.</td>
<td>• Code copies methods instead of arranging to invoke the originals.</td>
</tr>
<tr>
<td>• Or, mixed operations on different classes of numbers are implemented by arranging for the classes to share a common protocol.</td>
<td>• Code contains methods that appear to have been copied and modified.</td>
<td>• Code contains case analysis or a conditional that depends on the class of an object.</td>
</tr>
<tr>
<td>• Or, mixed operations on different classes of numbers are implemented by arranging for unconditional coercions.</td>
<td>• An object’s behavior is influenced by interrogating it to learn something about its class.</td>
<td></td>
</tr>
<tr>
<td>Exemplary</td>
<td>Satisfactory</td>
<td>Must Improve</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>