# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>2</td>
</tr>
<tr>
<td>Setup</td>
<td>2</td>
</tr>
<tr>
<td>Interactive debugging with μSmalltalk</td>
<td>2</td>
</tr>
<tr>
<td>Reading comprehension (10 percent)</td>
<td>3</td>
</tr>
<tr>
<td>Individual Problem</td>
<td>7</td>
</tr>
<tr>
<td>Pair Problems: Bignum arithmetic</td>
<td>8</td>
</tr>
<tr>
<td>Big picture of the solution</td>
<td>8</td>
</tr>
<tr>
<td>How to prepare your code for our testing</td>
<td>9</td>
</tr>
<tr>
<td>Unit testing bignums in Smalltalk</td>
<td>9</td>
</tr>
<tr>
<td>Using check-expect</td>
<td>9</td>
</tr>
<tr>
<td>Using check-print</td>
<td>10</td>
</tr>
<tr>
<td>Details of all the problems</td>
<td>10</td>
</tr>
<tr>
<td>A simple sanity check</td>
<td>16</td>
</tr>
<tr>
<td>More advice about testing natural numbers</td>
<td>17</td>
</tr>
<tr>
<td>Other hints and guidelines</td>
<td>19</td>
</tr>
<tr>
<td>Avoid common mistakes</td>
<td>19</td>
</tr>
<tr>
<td>Extra credit</td>
<td>20</td>
</tr>
<tr>
<td>What and how to submit: Individual problem</td>
<td>21</td>
</tr>
<tr>
<td>What and how to submit: Pair problems</td>
<td>22</td>
</tr>
<tr>
<td>How your work will be evaluated</td>
<td>22</td>
</tr>
</tbody>
</table>
Overview

Object-oriented programming has been popular since the 1990s, and like lambdas, object-oriented features are found everywhere. But the features are not always easy to tease out: many object-oriented languages, such as Java and C++, are hybrids, which mix objects with abstract data types or other notions of encapsulation and modularity. When you don’t already know how to program with objects, hybrid designs are more confusing than helpful. For that reason, we study pure objects, as popularized by Smalltalk: even simple algorithms send lots of messages back and forth among a cluster of cooperating, communicating objects. Popular languages that use similar models include Ruby, JavaScript, and Objective C.

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 μSmalltalk’s large initial basis.

• You implement bignums in μSmalltalk. 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 inheritance and pervasive dynamic dispatch that characterize idiomatic Smalltalk programs.

Setup

The μSmalltalk interpreter is in /comp/105/bin/usmalltalk. Many useful μSmalltalk sources are included the book’s git repository, which you can clone by

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

Sources that can be found in the examples directory includes copies of predefined classes, collection classes, shape classes, and other examples from the textbook.

Interactive debugging with μSmalltalk

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 826. 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 \texttt{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, (set &trace 0).

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

```
-> (new List)
standard input, line 3: Sending message (new) to class List
  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 object of class SmallInteger
    3 internal expression "(begin (print self) (print newline) self)" , line 1: (print 3<SmallInteger>) = 3<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\(^1\).

\(^1\)/cqs.small.txt
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 808:

```
(div: w 2)
(drawOn: shape self)
(do: shapes [block (shape) (drawOn: shape self)])
```

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 803e and 804, 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 820. 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 881. Now study the implementation of class Fraction, which starts around page 885.

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 of 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 875. 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 message 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 839. Look at the last part of the instance protocol for Number on page 838. 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 883.

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 10.11.1, which starts on page 929 (“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 10.7.6, which starts on page 881. 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 10.7.5, which starts on page 880. And read the section “laws for multiple dispatch” in the handout on “Program Design with Objects”\(^2\). Now, of the methods on class Number 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

\(^2\).../handouts/objproofs.pdf
• 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 work exercise 39(a) on page 948 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.

**39(a). Interfaces as Abstraction Barriers.** Do exercise 39(a) on page 948 of *Build, Prove, and Compare*. Put your solution in file frac-and-int.smt. Think about protocols, not implementation.

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 950:

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

This idiom enables you to change predefined classes without editing the source code of the μSmalltalk 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 839. Also in that section, read about Integer and Fraction.

• For the implementation of Integer, see page 883. 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 885. Study the implementation of method +, and observe how it relies on the exposure of representation through private methods num and den.

• Read the section “forms of data, access to representation”, which describes three levels of access, in the handout on “Program Design with Objects”.

• If nothing comes to you, try reading about how we get access to multiple representations in the object-oriented way: section 10.7.5, which starts on page 880. 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 work exercise 42 on page 949, exercise 43 on page 949, and exercise 44 on page 950 of Build, Prove, and Compare, and exercises T and ADT below.

Sometimes you want to do computations that require more precision than you have available in a machine word. Full Scheme, Smalltalk, Haskell, Icon, and many other languages provide “bignums,” which automatically expand to as much precision as you need. Unlike languages that use abstract data types, 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, and those messages are dispatched to lots of methods, which 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

3 ../handouts/objproofs.pdf
needs the big picture of which class is doing what. A good starting point is the Smalltalk bignums hand-

out.4

How to prepare your code for our testing

Many of our tests interact directly with the μSmalltalk interpreter. Our testing infrastructure enters defini-
tions 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

(method print ()
  (do: (decimal self) [block (digit) (print digit)]))

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. You have two options:

- Use check-expect with the decimal method.
- Use the new check-print form, which is not documented in the textbook.

Using check-expect

The semantics of check-expect are subtle, but the tests work about the way you would hope. You might remember that in μScheme, check-expect does not use the built-in = primitive—instead, it compares values uses something like the equal? function. This comparison enables μScheme’s check-expect to accept two different S-expressions that have the same structure. In much the same way, μSmalltalk does not use the built-in = method—instead, it compares objects using the similar: method. This comparison enables μSmalltalk’s check-expect to accept two different sequences that have the same elements. The details are likely to be important only if you have to debug a failing check-expect, but they can be found in section 10.3.5, which starts on page 821.

Using check-expect with the decimal method looks like this:

(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 ))

4://handouts/bignums.pdf
Because check-expect uses similar, it can accept a list of decimal digits returned by decimal as similar to (but not equal to) the array written using literal quote syntax.

Using check-print

Using check-expect is familiar, but aside from the obvious awkwardness of writing arrays of decimal digits, it has a more serious flaw: it can't detect bugs in your all-important print method. To help find bugs in print, I have added a check-print unit-test form, which is not yet documented in the book. The check-print form takes an expression and a literal result. The literal result must be a single token: either a name or a sequence of digits. Here are some example uses:

(check-print (x:to: Power 10 60) 1000000000000000000000000000000000000000000000000000000000000)
(check-print (x:to: Power 2 64) 18446744073709551616)
(check-print (negated (x:to: Power 2 64)) -18446744073709551616)
(check-print (* (new: Natural 256492) (new: Natural 666481)) 170947044652)

As soon as you have a print method working, I recommend using check-print.

Details of all the problems

42. Implementing arbitrary-precision natural numbers. Do exercise 42 on page 949 of Build, Prove, and Compare. Implement the protocol defined in Figure 10.17 on page 842. Put your solution in file bignum.smt.

The algorithms are the same algorithms you would have used on the sml assignment. So please do adapt your code from the sml assignment. Or if you prefer, you may adapt my solutions. (I provide both an array-based solution and a list-based solution, but the best solution, and the one that is easiest to adapt is the algebraic-datatype solution from the first ML homework.) Whatever code you adapt, be sure to attribute the source!

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

What representation should I use? You've got three choices:

- Array-based: Every natural number is represented by an array of digits.
- List-based (mutable): Every natural number is represented by a list of digits, where “list” means the mutable Smalltalk List abstraction.
- List-based (immutable): Every natural number is represented by a list of digits, where “list” means an immutable representation that is either empty or nonempty. In particular, every nonzero natural

---

5. sml.html
6. ../solutions/anatural.pdf
7. ../solutions/fnatural.pdf
8. ../solutions/ml.pdf
number is represented by an object of a special-purpose concrete class $\text{NatNonzero}$, which contains two instance variables. These instance variables hold the values of “self mod base” (the least significant digit) and “self div base” (all the other digits). A natural zero is represented by an instance of a different concrete class, also special-purpose [concrete class ‘NatZero], which represents only zero.

Between the ML warmup solutions⁹ and the ML modules solutions¹⁰, you can compare ML solutions that use different representations. 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. In particular, the immutable list in Smalltalk doesn’t look a lot like the immutable list in ML; good Smalltalk code most closely resembles the ML code that uses an algebraic data type¹¹.

The representations offer these tradeoffs:

- The solution based on an array is hardest to get right but easiest to get started. The easy part is that there is no case analysis in the data: every natural number is represented by an array. Hard parts include figuring out array lengths, managing array indices, and making sure not to confuse the index of a digit with the digit itself.
- The solution based on an immutable list, with special subclasses for empty and nonempty variants, is the easiest to get right but the hardest to get started. The easy part is that every method is specialized, so almost all of the case analysis is handled for you automatically, by method dispatch. The hard part is that you have to understand method dispatch.
- The solution based on a mutable List is possible in principle, but I don’t recommend it. Familiarity is nice, but in every other respect, I find it inferior to a solution based on an immutable list.

In my opinion, the superior representation is the immutable list. (As a bonus, this representation is also the best at helping you learn about programming with objects.) This representation makes it easy for me to understand my own code: I can look at any method, and I am confident that I understand exactly what it is doing. The array representation doesn’t give me that level of confidence.

**How must I document my representation?** As on the modules assignment, you must document the abstraction function and invariant for every concrete class. (A concrete class is one that is instantiated by sending it the new message, or by sending some other message whose method eventually sends new.)

- Document the abstraction function by writing, in a comment, an equation
  
  \[
  A (self) = \ldots
  \]
  
  and using instance variables on the right-hand side.

- Document the invariant either by writing a comment like
  
  \[
  I (self) = \ldots
  \]
  
  or by defining a private invariant method that answers a Boolean.

In each class, place your comments immediately below or to the right of the declaration of the instance variables.

---

⁹ ../solutions/index.html#ml
¹⁰ ../solutions/index.html#sml
¹¹ ../solutions/warmup.sml
How big is it? Using the hints in the book, I’ve written two implementations of class `Natural`:

- Using the array representation, my solution is about 120 lines of μSmalltalk code.
- Using the immutable list representation, my solution is about 150 lines of μSmalltalk code.

I have not written a solution that uses mutable lists.

Related reading:

- There is a detailed implementation guide in the bignums handout. It discusses 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 10.7.5, which starts on page 880.
- In the handout on “Program Design with Objects”, read the section on how the design steps are adapted for use with objects. Focus on steps 6, 7, and 8: algebraic laws, case analysis, and results. In the same handout, you may also wish to revisit the three levels of access to representation. You will need level C, but there is no need for double dispatch here.
- 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 881.
- For the interface to a Smalltalk array, study the `Collection` protocol in section 10.4.5, which starts on page 829. You have access to the protocol in Figure 10.10 on page 832, but you are more likely to use the `KeyedCollection` protocol in Figure 10.11 on page 834, especially at: and at:put:. Don’t overlook the `Arrays` section on pages 879 and 880, 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 836.
- For abstraction functions and invariants, you may wish to revisit the handout on design with abstract data types.

43. Implementing arbitrary-precision integers. Do exercise 43 on page 949 of Build, Prove, and Compare. Add your solution to file `bignum.smt`, following your solution to exercise 42.

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 must I document my representation? As on the previous exercise, you must document the abstraction function and invariant for every concrete class. For example if you follow the guidelines

---

12./handouts/bignums.pdf
13./handouts/objproofs.pdf
14Note: an object of class `Natural` is not a `Number` as Smalltalk understands it. In particular, class `Natural` does not support methods `negated` or `reciprocal`.
15./handouts/adtproofs.pdf
and define an *abstract* class `LargeInteger` with two concrete subclasses `LargePositiveInteger` and `LargeNegativeInteger`, you’ll need to document only the two concrete subclasses.

**How big is it?** My solutions for the large-integer classes are 30 lines apiece.

**Related reading:** This problem is all about dynamic dispatch, including double dispatch.

- Read section 10.7.5, which starts on page 880.
- Read the last section, “Laws for double dispatch,” in the handout on “Program Design with Objects”\(^\text{16}\).

You’ll also have a chance to practice double dispatch in recitation.

**Helpful code.** I have no wish to torture anyone with the details of signed-integer division. For those of you who might not wish to translate the nested conditionals found in the `divide` function in the \(\mu\)Scheme chapter, here are some methods I have defined on my own large-integer classes.

- On class `LargeInteger`:
  ```scheme
  (method div: (n) (sdiv: self n))
  ``

- On class `LargePositiveInteger`:
  ```scheme
  (method sdiv: (anInteger)
    (ifTrue:ifFalse: (strictlyPositive anInteger)
      {{(withMagnitude: LargePositiveInteger (sdiv: magnitude anInteger))}
       {{(negated (sdiv: (- (- self (new: LargeInteger anInteger))
                  (new: LargeInteger 1))
               (negated anInteger))))}})
  ``

- On class `LargeNegativeInteger`:
  ```scheme
  (method sdiv: (anInteger)
    (ifTrue:ifFalse: (strictlyPositive anInteger)
      {{(negated (sdiv: (- (+ (negated self) (new: LargeInteger anInteger))
                  (new: LargeInteger 1))
               anInteger))}
       {{(sdiv: (negated self) (negated anInteger))))})
  ``

I also don’t wish to torture anyone with two’s-complement representations. The following code on class `LargeInteger` will ensure that the `negated` method defined on `SmallInteger` does not overflow:

```scheme
(class-method new: (anInteger)
  (ifTrue:ifFalse: (negative anInteger)
    {{(negated (+ (new: self 1) (new: self (negated (+ anInteger 1)))))}
     {{(magnitude: (new LargePositiveInteger) (new: Natural anInteger))})}})
```

44. **Modifying `SmallInteger` so operations that overflow roll over to infinite precision.** Do exercise 44 on page 950 of *Build, Prove, and Compare*. Put your solution in a fresh file, `mixnum.smt`. On the first line of file `mixnum.smt`, include your other solutions by writing `(use bignum.smt)`\(^\text{17}\).

\(^{16}\)../handouts/objproofs.pdf

\(^{17}\)If there is a bug in your solution to exercise 44, it can break your solutions to the previous exercises. By putting the solution to exercise 44 in its own file, we make it possible to test your other code independently.
You must modify SmallInteger without editing the source code of the \\Smalltalk interpreter. To do so, you will redefine class SmallInteger using the idiom on page 950:

\[
\begin{align*}
\text{(class SmallInteger SmallInteger} \\
\text{()} \\
\text{... new method definitions ...} \\
\text{)}
\end{align*}
\]

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, especially the example in the handout. 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 827.
- Read the description of \texttt{at:ifAbsent:} in the keyed-collection protocol in Figure 10.11 on page 834. Now study this expression:
  \[
  (\texttt{at:ifAbsent: } ' (0 1 2) 99 \{0\})
  \]
  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, \texttt{at:ifAbsent:} sends value to the “exception block” \(\{0\}\), which ultimately answers zero.
- Study the implementation of the \texttt{at:} method in \{other methods of class KeyedCollection 871d\}, which uses \texttt{at:ifAbsent:} with an “exception block” that causes a run-time error if value is sent to it.
- Finally, study the overflow-detecting primitive methods in exercise 44 on page 950, and study the implementation of \texttt{addSmallIntegerTo:} in the code chunk immediately below. That is the technique you must emulate.

**T. Testing Bignums.** In standalone file \texttt{bigtests.smt}, you will write 9 tests for bignums:

- 3 tests will test only class \texttt{Natural}.
- 3 tests will test the large-integer classes, which are built on top of class \texttt{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 \textbf{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 checked using the check-print form described above.

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-print (run: Test10Power 10) 10000000000)

; Summary: 10 to the 30th power, mixed arithmetic
(check-print (run: Test10Power 30) 1000000000000000000000000000000)

Here is another complete example:

; Summary: 20 factorial
(define factorial (n)
   (ifTrue:ifFalse: (strictlyPositive n)
      {(* n (value factorial (- n 1)))}
      {1}))
)
(check-print (value factorial 20) 2432902008176640000)

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

ADT. Collect representations, abstraction functions, and invariants.

In a new file adt.txt, summarize your design work:

- For each concrete class you defined to represent natural numbers in exercise 42,
  - List the instance variables.
  - Copy and paste the class's abstraction function and invariant.

- For the natural numbers, explain in one or two sentences why you chose the representation that you did. In addition, please tell us what went well and if you have any regrets.

- For each concrete class you defined to represent large integers in exercise 43,
A simple sanity check

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

- These tests only ever multiply 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 a more general version of power\(^{18}\):

```scheme
(class Power Object

(class-method x:to: (x n) [locals half]
 (ifTrue:ifFalse: (= n 0)
  { (coerce: x 1) }
  { (set half (x:to: self x (div: n 2)))
    (set half (* half half))
    (ifTrue: (= (mod: n 2) 1)
     { (set half (* x half)))
    half) })

(check-expect (x:to: Power 2 3) 8)
(check-expect (x:to: Power 3 4) 81)
(check-expect (x:to: Power (/ 1 3) 3) (/ 1 27))

And here is code that computes and prints factorials\(^{19}\):

```scheme
(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:

1! = 1
2! = 2
3! = 6
4! = 24
5! = 120

\(^{18}\)./power.smt
\(^{19}\)./factorial.smt
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

This computation will exhaust the “CPU fuel” used to recover from infinite loops—so you will need to test it while running

env BPCOPTIONS=nothrottle usmalltalk

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:

```
(define nat-test (ns) ; alert! µScheme code
  (letrec ([exp-of (lambda (ns)
      (if (null? ns)
        0
        (list3 '+ (car ns) (list3 '* 10 (exp-of (cdr ns))))))])
    (let* ([left (lambda () (printu 40))]
      [right (lambda () (printu 41))]
      [space (lambda () (printu 32))]
      [_ (left)]
      [_ (print 'check-print)]
      [_ (space)]
      [_ (print (exp-of (reverse ns)))]
    )
  )
)
```

30 ./nat-test.scm
For example,

-> (nat-test '(1 2 3))
(check-print (+ 3 (* 10 (+ 2 (* 10 (+ 1 (* 10 0))))) 123)
Printed!

If you don’t have multiplication working yet, you can use the following class\(^{21}\) to multiply by 10:

```
(class Times10 Object
  ()
  (class-method by: (n) (locals p)
     (set p n) ; p == n
     (set p (+ p p)) ; p == 2n
     (set p (+ p p)) ; p == 4n
     (set p (+ p p)) ; p == 5n
     (set p (+ p p)) ; p == 10n
     p))
```

This idea will test only your addition; if you have bugs there, fix them before you go on.

You can write, in $\mu$Smalltalk instead of $\mu$Scheme, 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.

Here’s an example of using `ghci` to generate numbers for testing:

\(^{21}\).`times10.smt`
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\(^\text{22}\) 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\(^\text{23}\) 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\(^\text{24}\) (see also his distribution\(^\text{25}\)). 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.

- In Hanson’s code, `XP_add` does add with carry. `XP_sub` does subtract with borrow. `XP_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 \(\mu\)Smalltalk.
- Hanson’s implementation uses mutation extensively, 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 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 a **terrible mistake** to make decisions by interrogating an object about its class—a so-called “run-time type test.” Run-time type tests destroy behavioral subtyping. This mistake is most commonly made in two places:

- If you are representing a `Natural` number as a list of digits, you may be tempted to interrogate the representation to ask “are you nil or cons?” This is the functional way of programming, but

\[^{22}\] ../handouts/bignums.pdf
\[^{23}\] ../readings/indexbody.html#cii
\[^{24}\] xp.c
\[^{25}\] http://www.cs.princeton.edu/software/cii
in Smalltalk, it is wrong. You must make the decision by sending a message to an object, and the method that is dispatched to will know whether it is nil or cons.

- If you are mixing arithmetic on large and small integers or on integers and fractions, you may be tempted to interrogate an argument about its class. This interrogation is wrong. You must instead figure out how to accomplish your goals by sending messages to the argument—probably including messages from some private protocol.

There is a right way to do case analysis over representations: entirely by sending messages. For an example, study how we calculate the length of a list: we send the size message to the list instance. Method size is dispatched to class Collection, where it is implemented by using a basic iterator: the do: method. If you study the implementation of do: on classes Cons and ListSentinel (which terminates a μSmalltalk list), you’ll see the case analysis is done by the method dispatch:

- Sending do: to a cons cell iterates over the car and cdr.
- Sending do: to a sentinel does nothing (thereby terminating the iteration).

The idea of “case analysis by sending messages” applies equally well to arithmetic—and the suggestions in the bignums handout are intended to steer you in the right direction. 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 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 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 μSmalltalk is written using S-expressions, you could consider writing this program in μScheme—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.

Speed variations. 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. And measure the time needed to compute the first 50 Catalan numbers.

   The Catalan numbers, which make better test cases than factorial, are defined by these equations (from Wikipedia):

   \[
   C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i \ C_{n-i} \quad \text{for} \ n \geq 0,
   \]

---

26. ../handouts/bignums.pdf
\[C_0 = 1 \quad C_{n+1} = \sum_{i=0}^{n} C_i \cdot C_{n-i}\]

2. Determine the largest possible base that is still a power of 10. Explain your reasoning. Change your class to use that base internally. **Measure** the time needed to compute the first 50 factorials, and also the time needed to compute the first 50 Catalan numbers.

3. In both cases, measure the additional time required to **print** the numbers you have computed.

4. Specialize your \( b = 10 \) code so that the decimal method works by simply reading off the decimal digits, without any short division. **Measure** the improvement in printing speed.

5. Finally, try a compromise, like \( b = 1000 \), which should use another specialized decimal method, making both arithmetic and decimal conversion reasonably fast. Can this implementation beat the others?

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\(^{27}\) 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:

- 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.,

\(^{27}\) readings/indexbody.html#division

21
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 exercises you worked (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 and 43. 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 mixnum.smt showing your solution to exercise 44. This file should incorporate your other solution by reference, using the line
  
  (use bignum.smt)
  
  at the beginning. Do not duplicate code from bignum.smt.
- A file bigtests.smt containing your solution to exercise T.
- A file adt.text containing your solution to exercise ADT.

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</td>
<td>• The base used for natural numbers appears in exactly one place, but code that</td>
<td>• The base used for natural numbers appears in multiple places.</td>
</tr>
<tr>
<td>code that depends on it consults that place.</td>
<td>depends on it knows what it is, and that code will break if the base is</td>
<td>• Overflow is detected only by assuming the number of bits used to represent a</td>
</tr>
<tr>
<td>• Or, the base used for natural numbers appears in exactly one place, and</td>
<td>changed in any way.</td>
<td>machine integer, and the number of bits is <em>implicit</em> in the value of some</td>
</tr>
<tr>
<td>code that depends on either consults that place or assumes that the base</td>
<td>• Overflow is detected only by assuming the number of bits used to represent a</td>
<td>frightening decimal literal.</td>
</tr>
<tr>
<td>is some power of 10</td>
<td>machine integer, but the number of bits is <em>explicit</em> in the code.</td>
<td>• Code contains more than one avoidable conditional.</td>
</tr>
<tr>
<td>• No matter how many bits are used to represent a machine integer,</td>
<td>• Code contains one avoidable conditional.</td>
<td>• Mixed operations on different classes of integers are implemented by</td>
</tr>
<tr>
<td>overflow is detected by using appropriate primitive methods, not by</td>
<td>• Mixed operations on different classes of integers involve explicit</td>
<td>interrogating objects about their classes.</td>
</tr>
<tr>
<td>comparing against particular integers.</td>
<td>conditionals.</td>
<td>• Code copies methods instead of arranging to invoke the originals.</td>
</tr>
<tr>
<td>• Code uses method dispatch instead of conditionals.</td>
<td>• Code protects itself against exceptional or unusual conditions by using</td>
<td>• Code contains case analysis or a conditional that depends on the class of an</td>
</tr>
<tr>
<td>• Mixed operations on different classes of numbers are implemented using</td>
<td>Booleans.</td>
<td>object.</td>
</tr>
<tr>
<td>double dispatch.</td>
<td>• Code contains methods that appear to have been copied and modified.</td>
<td>• An object’s behavior is influenced by interrogating it to learn something</td>
</tr>
<tr>
<td>• Or, mixed operations on different classes of numbers are implemented</td>
<td>• An object’s behavior is influenced by interrogating it to learn something</td>
<td>about its class.</td>
</tr>
<tr>
<td>by arranging the classes to share a common protocol.</td>
<td>about its class.</td>
<td></td>
</tr>
<tr>
<td>• Or, mixed operations on different classes of numbers are implemented</td>
<td>• Code contains methods that appear to have been copied and modified.</td>
<td></td>
</tr>
<tr>
<td>by arranging for unconditional coercions.</td>
<td>• An object’s behavior is influenced by interrogating it to learn something</td>
<td></td>
</tr>
<tr>
<td>• Code deals with exceptional or unusual conditions by passing a suitable</td>
<td>about its class.</td>
<td></td>
</tr>
<tr>
<td><em>exnBlock</em> or other block.</td>
<td>• Code contains methods that appear to have been copied and modified.</td>
<td></td>
</tr>
<tr>
<td>• Code achieves new functionality by reusing existing methods, e.g., by</td>
<td>• An object’s behavior is influenced by interrogating it to learn something</td>
<td></td>
</tr>
<tr>
<td>sending messages to <em>super</em>.</td>
<td>about its class.</td>
<td></td>
</tr>
<tr>
<td>• Or, code achieves new functionality by adding new methods to old classes</td>
<td>• Code contains methods that appear to have been copied and modified.</td>
<td></td>
</tr>
<tr>
<td>to respond to an existing protocol.</td>
<td>• An object’s behavior is influenced by interrogating it to learn something</td>
<td></td>
</tr>
<tr>
<td>• An object’s behavior is controlled by dispatching (or double dispatching) to</td>
<td>about its class.</td>
<td></td>
</tr>
<tr>
<td>a method.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>Exemplary</td>
<td>Satisfactory</td>
<td>Must Improve</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
</tbody>
</table>
