6.005 Elements of Software Construction
Fall 2008

6.005 Elements of Software Construction | Fall 2008
Problem Set 2: The Symbolic Paradigm

The purpose of this problem set is to give you practice in the basic techniques of the symbolic paradigm. You will model datatype productions, write recursive functions over those datatypes, practice with structural induction, and implement datatypes as classes.

Datatype Reasoning

Length of append [10%]. Given a datatype `List` and two functions over the datatype `size` and append defined as follows:

```
List<E> = Empty + Cons(first:E, rest:List<E>)

size: List<E> → int
   size(Empty) = 0
   size(Cons(f,r)) = 1 + size(r)

append: List<E> x List<E> → List<E>
   append(Empty, l) = l
   append(Cons(e,l1), l2) = Cons(e, append(l1,l2))
```

Show using structural induction that `size(append(x,y)) = size(x) + size(y)`.

Datatype Modeling

Suppose you're developing a system for doing physics calculations (simple mechanics, as used in 8.01; e.g. Newton's laws of motion and Newton's law of gravitation.).

Physical units [10%]. One helpful feature in a physics-handling system is to check expressions for unit correctness. For example, an expression that adds mass to velocity is nonsensical. Model the units of a physical quantity used in mechanics. Your datatype should be able to handle dimensionless values (like *pi* and *e*), the basic SI units for mechanics (meters, grams, and seconds), and the units of physical variables like velocity, momentum,

force, and energy.

Expressions for physical computations [10%]. Design a datatype that models the mathematical expressions you need to represent problems in mechanics. The expressions should be over the real numbers; should be able to represent numerical constants (like 0), named constants (like G or g), and variables (like F, m, a); should include the basic arithmetic operators (addition, multiplication, exponentiation, etc.), differentiation and integration in one variable, and equality (like F=ma). Constants and variables should have physical units, using the datatype you defined in the previous problem. **Briefly discuss** the design decisions you made -- e.g., does your datatype permit nonsensical expressions?

Note: These datatypes should be defined using abstract datatype productions (similar to the way we defined List and Tree in class), **not in Java**.

Datatype Operations

Unit checking [15%]. Define a boolean-valued function `unitsCheck` over your physical expressions that is true if and only if the the units of the expression check correctly. Write your function as a recursive definition, **not in Java**. Hints:

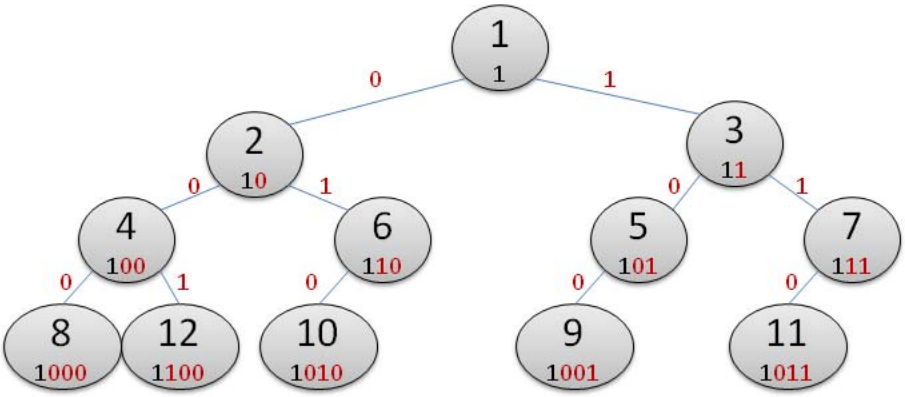
- You may need to define additional helper functions to determine the units of an expression or test for unit equality. Don't just assume the existence of these helper functions; provide definitions for them if you need them.
- Checking for unit equality may be tricky, depending on how you defined your units datatype. It may help to use a canonical form that puts a units expression in lowest terms and determines the exponents of each of the three physical quantities (mass, length, time). For example, the units of acceleration (m/s^2) would be canonically represented by length 1, time -2, mass 0. You can define the canonical form either as a new datatype (i.e., a triple of integers) or as a set of functions over your units datatype (e.g., mass:Units → int).

Arrays [15%]. This problem uses an immutable datatype to represent arrays, which are data structures that map integers to objects. An array is represented by a binary tree, in which each node stores an element of the array:

```
Array<E> = Empty + Node(elem:E, left:Array<E>, right:Array<E>)
```

The array is indexed starting from 1. Each tree node has an index given (in binary) by its path to the root, as follows. Start with 1 for the node itself, and append 0 or 1 for each step back to the root, depending on whether it was a left branch or a right branch. The diagram

below shows the nodes with index 1 through 12.



The element at index *i* is stored in the tree node with index *i* according to this scheme, so a [1] would be stored in the root. Note that indexes are *not* stored in the tree; an index is used to find a node by walking down from the root. Also, an array may not have gaps, so if it contains *n* elements, then the elements must be indexed from 1 to *n*. The binary tree is not necessarily complete, however, so some nodes in the tree may have only one nonempty child (like nodes 5, 6, and 7 in the picture above).

Give recursive definitions for the following operations:

- *size(a)* returns the number of elements in an array.
- *get(a,i)* returns the element at index *i* in array *a* (or is undefined if *i* is out of range).
- *put(a,i,e)* returns a new array which maps index *i* to element *e* and is otherwise identical to *a*. (Undefined if *i* is negative or zero, or if it would create an array with a gap.)

Hint: you may need to write conditional expressions in your function definitions. Here's a simple model:

```
abs: int → int
   abs(x) = if x < 0 then -x else x
```

Implementing Datatypes

Arrays [20%]. Implement the array datatype from the previous problem using Java classes, and implement `get`, `put`, and `size` as methods. Write a JUnit test suite that tests your datatype. Note that your datatype should be immutable, and you will have to make design decisions about what the methods should do when the result is undefined.

Arbitrary-precision rational numbers [20%]. Implement an immutable abstract data type for rational numbers. Represent a rational number using two java.math.BigInteger values, a numerator and a denominator, which should always be stored in lowest form (i.e., not 4/8, but 1/2). You'll find BigInteger's gcd operation useful; a.gcd(b) returns the greatest common divisor of a and b.

Implement your data type as a Java class called Rat. It should have the following parts:

- Representation, consisting of private fields
- Rep invariant, implemented by a checkRep() method
- Abstraction function, implemented by a toString() method
- Constructor Rat(BigInteger num, BigInteger denom)
- Operations provided by the following methods:
  public boolean isZero()
  public boolean isOne()
  public Rat plus(Rat that)
  public Rat minus(Rat that)
  public Rat times(Rat that)
  public Rat divide(Rat that)
- Preconditions and postconditions on the constructor and operations
- A JUnit test suite called RatTest that tests Rat

Infrastructure

No code is provided for this problem set. A directory called `pset2` will be created for you in your personal repository, containing a copy of this file. In addition to your code and test cases, you should commit your solutions to the exercises as a single PDF file.