Formal Specification of Abstract Datatypes
Exercise 3 (June 1)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- a PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - the formal specifications in the style of "Thinking Programs",
  - the CafeOBJ specifications,
  - comprehensive tests of the CafeOBJ specifications (sample reductions),
  - optionally any explanations or comments you would like to make;
- the CafeOBJ (.mod) file(s) of the specifications.

Exercise: The Integers

From Wikipedia[1]: ". . . The set of *integers* consists of *zero* (0), the *positive natural numbers* (1, 2, 3, . . .), also called *whole numbers* or *counting numbers* and their additive inverses (the *negative integers*, i.e., -1, -2, -3, . . .). . . . "

Formal Specification Develop a formal specification of the domain of integers with the usual operations in the style presented in the course:

```
spec INTEGER import NAT and BOOL :=
free {
  type Int = ...
  ...
}
then ...
{
  const 0I: Int
  const 1I: Int
  fun +: Int × Int → Int
  fun -: Int × Int → Int
  fun -: Int → Int
  fun *: Int × Int → Int
  pred ≤ ⊆ Int × Int
  pred < ⊆ Int × Int
  ...
}
```

Actually, develop *two* specifications, each with a different "representation" of integers:

- In the first version, use the representation of an integer as a pair of a natural number and a sign (represented by a boolean value)

  ```
  free {
    type Int = i(Bool,Nat)
    ...
  }
  ```

  but make sure that 0 is uniquely represented ($+0 = -0$).

- In the second version, use the representation of an integer as a difference of two natural numbers

  ```
  free {
    type Int = i(Nat,Nat)
    ...
  }
  ```

  but make sure that every integer is uniquely represented ($5 - 3 = 4 - 2 = \ldots = 2 - 0$).

[1] https://en.wikipedia.org/wiki/Integer

On top of such a free core specification of the integer domain, specify all integer operations either loosely or freely (as you prefer, but do not forget the keyword “free” in the later case).

Is your specification monomorphic? Justify your answer.

CafeOBJ Implement *one* of the specifications developed above in CafeOBJ by a tight module:

```
module! INTEGER
{
  protecting (NAT)
  signature
  {
    [ Int ]
    ...
  }
  axioms
  {
    ...
  }
}
```

Test the CafeOBJ module with several reductions. Give the input and output of each test and your interpretation of the results (do they indicate errors in your specification or not?). If your specification contains errors, use the trace facilities of CafeOBJ for debugging.

Does your CafeOBJ module denote the same datatype as the one of the previous specification? Justify your answer.