COL226: Programming Languages
II semester 2022-23

Assignment: The Big `Rational` Package

Introduction

Anybody who says that modern digital computers implement real numbers is actually lying. However it is possible to implement arbitrary-precision rational numbers.

A **rational** number (in standard form) is an ordered pair (p, q) usually written as "p/q" where

- p is an integer, called the **numerator**
- $q > 0$ is a positive integer, called the **denominator**.

The rational number is in **fractional-normal form** if $|p|$ and q are relatively prime i.e. $gcd(|p|, q) = 1$.

Rational numbers may also be represented in decimal as given by the regular expression

$$[S]I.N(R)$$

where

- $\{0,1,2,3,4,5,6,7,8,9,.,(,),+,\tilde{}\}$ is the alphabet of the regular expression,
- S is an optional sign "`+`" or "`~`",
- $d \in \{0,1,2,3,4,5,6,7,8,9\}$,
- I is a possibly empty integer part defined by "d^*",
- "." is the decimal point,
- N defined by "d^*" is the *non-recurring fractional part*,
- R defined by "d^+" is the *recurring fractional part* enclosed in the pair of parentheses "(" and ")".
- The decimal representation is in **decimal-normal** form iff N is the shortest possible string of non-recurring digits. Hence
  - the fraction $\frac{1}{3}$ is represented either as .(3) or +.(3). Equivalent forms such as .3(3) or .33(3) is not decimal-normal forms..
  - $3\frac{1}{3} = \frac{10}{3}$ would be written as `3.(3)` in decimal-normal form.
  - the fraction $\frac{1}{7}$ in decimal-normal form is written as .(142857); other equivalent forms such as .142(85714) are not decimal-normal. However $\frac{10}{7}$ is written as `1.(428571)` in decimal-normal form.

Consider the rational numbers with the following signature.

```
signature RATIONAL =
  sig
    type rational
    exception rat_error
    val make_rat: bigint * bigint -> rational option
    val rat: bigint -> rational option
    val reci: bigint -> rational option
    val neg: rational -> rational
    val inverse : rational -> rational option
    val equal : rational * rational -> bool (* equality *)
    val less : rational * rational -> bool (* less than *)
    val add : rational * rational -> rational (* addition *)
    val subtract : rational * rational -> rational (* subtraction *)
    val multiply  : rational * rational -> rational (* multiplication *)
    val divide : rational * rational -> rational option (* division *)
    val showRat : rational -> string
    val showDecimal : rational -> string
    val fromDecimal : string -> rational
    val toDecimal : rational -> string
  end;
```

where

- `showRat` displays a rational number in *fractional-normal* form.
- `showDecimal` displays the rational number in *decimal-normal* form.
- `bigint` is an integer of arbitrary size (not limited by the implementation) and could have a magnitude much larger than the `maxInt` or much smaller than the `minInt` defined by the SML-NJ structure `Int`.
- `fromDecimal` may take a possibly non-standard decimal form and create a rational from it.
- `toDecimal` converts a rational number into its standard decimal representation.
- `rat` inputs an integer i and converts it into the rational $\frac{i}{1}$.
- `inverse` finds the reciprocal of a non-zero rational.
- `reci` finds the reciprocal of a non-zero integer.
- `make_rat` takes any two integers and creates a rational number fractional-normal form. Hence `make_rat (4, ~10) = (~2,5)`.
- `neg` takes any rational number and negates it.

What you need to do

Your main job is to make the SML-NJ command-line interface suitable for calculations using rational numbers.

1. Design a grammar for **rational numbers** which seamlessly allows large integers to be treated as rationals, rationals expressed as numerator-denominator pairs of large integers, and rational numbers in decimal form. Negative rational numbers are written $\tilde{}p/q$ where p and q are large non-negative and large positive integers respectively, following the SML-NJ convention for negative integers and negative reals. *Document this grammar in your* `README.md` *file.*
2. Design a grammar for **rational number expressions** over the above grammar for rational numbers. The binary operators are the usual "`+`", "`-`", "`*`", and "`/`" in infix form with the usual rules of associativity and precedence. The expressions may contain variables of type `rational` or `int` and may be mixtures of rational numbers of the form p/q (where p and q are large integers) as well as rational numbers in decimal form (as defined above). *Document this grammar in your* `README.md` *file.*

3. Design a (command-line) user-interface for rational number expressions. You may use `ML-Lex`, `ML-Yacc` etc to scan and parse the expressions with the usual rules of associativity and precedence. *Document all design and pragmatic decisions in your* `README.md` *file.*
4. In order to be able to deal with rationals involving large integers first design a signature `BIGINT` for integers of type `bigint` and a structure `BigInt: BIGINT` which implements all the integer functions involving integers of unbounded magnitude.
5. Implement the structure

   ```
   structure Rational :  RATIONAL
   ```

   as a **functor** of the `BIGINT` package.
6. It should be possible to start SML-NJ with the command prompt

   ```
   $ sml rational.sml
   Standard ML of New Jersey v110.79 [built: Sat Oct 26 12:27:04 2019]
   [opening rational.sml]
   ```

   and proceed to use the rational number calculator with rationals with arbitrarily large representations without suffering overflows and underflows, unless the entire memory allocated to it is insufficient and the heap is full.
7. All rationals are displayed in their **normal** forms i.e. either *fractional-normal* form (which is the default) or *decimal-normal* form (if required by the user).

Note for all assignments in general

1. Some instructions here may be overridden explicitly in the assignment for specific assignments
2. Upload and submit a single zip file called `<entryno>.zip` where `entryno` is your entry number.
3. You are *not* allowed to change any of the names or types given in the specification/signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
4. The evaluator may use automatic scripts to evaluate the assignments (especially when the number of submissions is large) and penalise you for deviating from the instructions.
5. You may define any new auxiliary functions/predicates if you like in your code besides those mentioned in the specification.
6. Your program should implement the given specifications/signature.
7. You need to think of the *most efficient way* of implementing the various functions given in the specification/signature so that the function results satisfy their definitions and properties.
8. In a large class or in a large assignment, it is not always possible to specify every single design detail and clear each and every doubt. So you are encouraged to also include a `README.txt` or `README.md` file containing
   - all the decisions (they could be design decisions or resolution of ambiguities present in the assignment) that you have taken in order to solve the problem. Whether your decision is "reasonable" will be evaluated by the evaluator of the assignment.
   - a description of which code you have "borrowed" from various sources, along with the identity of the source.

- all sources that you have consulted in solving the assignment.
- name changes or signature changes if any, along with a full justification of why that was necessary.

9. The evaluator may look at your source code before evaluating it, you must explain your algorithms in the form of comments, so that the evaluator can understand what you have implemented.
10. Do *not* add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
11. There is a serious penalty for code similarity (similarity goes much deeper than variable names, indentation and line numbering). If it is felt that there is too much similarity in the code between any two persons, then both are going to be penalized equally. So please set permissions on your directories, so that others have no access to your programs.
12. To reduce penalties, a clear section called "**Acknowledgements**" giving a detailed list of what you copied from where or whom may be be included in the `README.txt` or `README.md` file.