ADS for Rational Numbers

An application program which uses rational numbers needs to perform operations on these numbers without any mention of how they are implemented. Here is a set of functions which implement rational numbers. Note that the *Constructors* and *Accessors* provide the abstraction barrier. *Recognizers*, *Functions* and *Invariants* define the structure.

REPRESENTATION

Let rational numbers be represented by the set of labels `{r,r1,r2,...}`.

A rational number is defined by the division of two integers. For any rational number **`ri`**, let the numerator be represented as **`ni`** and the denominator be **`di`**.

We will assume that the class *integer* is defined.

RECOGNIZERS

```
is-ratnum[x]  = True          iff    x is a rational number
is-denom[x]   = True          iff    x is an integer number
is-numer[x]   = True          iff    x is an integer number

is-zero[r]    = True          iff    get-numer[r] = 0
is-one[r]     = True          iff    get-numer[r] = get-denom[r]
is-illegal[r] = True          iff    get-denom[r] = 0
```

Note that `is-denom` and `is-numer` are typing predicates. They return True for any integer `x`. We arbitrarily elect to restrict numerators and denominators to positive integers, associating the sign of a rational number with the rational number itself. To express these facts:

```
is-denom[x] = True      iff
            (is-integer[x] and not[is-negative[x]] and not[x=0])

is-numer[x] = True      iff    (is-integer[x] and not[is-negative[x]])
```

Specific denominators and numerators can be recognized by the following relations:

```
is-denom-of[x,r] = True       iff    x is the denominator of rational number r
is-numer-of[x,r] = True       iff    x is the numerator of rational number r
```

Example:

```
is-ratnum[r]  =def=
  let n = get-numer[r]
      d = get-denom[r]
    if (is-denom[d] and is-numer[n])
      then return True
      else return False
```

CONSTRUCTORS

`make-ratnum[sign,numer,denom]`	returns the rational number `r = numer/denom`
`destroy-ratnum[r]`	frees memory associated with ratnum `r`

These functions transfer between the abstraction and the implementation choices. See the implementation section.

ACCESSORS

`get-numer[r]`	returns the numerator of `r`
`get-denom[r]`	returns the denominator of `r`
`get-sign[r]`	returns the sign of the ratnum `r`

These functions also transfer between abstraction and implementation.

FUNCTIONS

`equal-ratnum[r1,r2] = True`	iff `r1 = r2`
`add-ratnum[r1,r2]`	returns `r3 = r1+r2`
`sub-ratnum[r1,r2]`	returns `r3 = r1-r2`
`mult-ratnum[r1,r2]`	returns `r3 = r1*r2`
`div-ratnum[r1,r2]`	returns `r3 = r1/r2`
`reduce-ratnum[r]`	returns a reduced ratnum
`print-ratnum[r]`	returns <the graphical representation of a ratnum>
`ratnum-to-decimal[r]`	returns the decimal value of `r`

The code for these functions does not rely on the implementation of ratnums. Some pseudocode examples:

```
equal-ratnum[r1,r2]  =def=
  let n1 = get-numer[r1]
      n2 = get-numer[r2]
      d1 = get-denom[r1]
      d2 = get-denom[r2]
      res = ((n1*d2) = (n2*d1))          ;here * is integer multiply
    return res                           ;res is a Boolean equality test
```

The following code assumes that the two ratnums are positive, it would be a little more complex if signed ratnums were being added.

```
add-ratnum[r1,r2]  =def=
  let n1 = get-numer[r1]
      n2 = get-numer[r2]
      d1 = get-denom[r1]
      d2 = get-denom[r2]
      n3 = ((n1*d2) + (n2*d1))
      d3 = (d1*d2)
    return make-ratnum[n3,d3]
```

INVARIANTS

Invariants provide pre and post tests for ratnum computations. The appropriate invariants should be asserted as error checks before and after programs which use any rational numbers.

```
is-ratnum[r] or not[is-ratnum[r]] = True

is-integer[get-denom[r]]   = True
is-integer[get-numer[r]]   = True
not[equal[get-denom[r],0]] = True
<many other possible invariant equations>
```

Example of an embedded pre-test:

```
greater-than-one-ratnum[r]  =def=
  if is-ratnum[r]
    let n = get-numer[r]
        d = get-denom[r]
        comp = (n>d)
      return comp
    else return error-signal[r,"not-ratnum"]
```

AXIOMS

Axioms are another kind of invariant, defining the numerical behavior of ratnums. An infinite number of theorems can be derived from a set of axioms. Only a very few will be of utility. Finding essential and important theorems is part of the task of domain modeling.

`r1+r2 = r2+r1`	commutativity of addition
`r1+(r2+r3) = (r1+r2)+r3`	associativity of addition
`r1+0 = r1`	additive identity
`r1*r2 = r2*r1`	commutativity of multiplication
`r1*(r2*r3) = (r1*r2)*r3`	associativity of multiplication
`r1*1 = r1`	multiplicative identity
`if d=1 then r=n`	1 denominator
`if n=0 then r=0`	0 numerator
`x+(n/d) = ((x*d)+n)/d`	adding an integer
`(n1/d1)+(n2/d2) = ((n1*d2)+(n2*d1))/(d1*d2)`	adding two ratnums
`x*(n/d) = (x*n)/d`	multiplying by an integer
`(n1/d1)*(n2/d2) = (n1*n2)/(d1*d2)`	multiplying two ratnums

IMPLEMENTATION

Let's implement ratnums as 3-tuples, that is as triples consisting of a sign bit and two integers. We first elect to implement 3-tuples as a list of three integers, encoding the sign as `0=-` and `1=+`:

```
make-ratnum[s,n,d] =def=
  return List[s,n,d]

get-sign[r] =def=
  return First[r]

get-numer[r] =def=
  return Second[r]

get-denom[r] =def=
  return Third[r]
```

In the above, `get-sign[r]` is in error, since it will return 0 or 1, not the “sign” of the ratnum. An apparent fix is:

```
get-sign[r] =def=
  if First[r] = 0
    then return “+”
    else if First[r] = 1
      then return “-“
      else return Error
```

The problem now is that the return of a function name (+ or -) requires either a string encoding, thus changing the intended type of the sign data structure, or it requires language support for returning functions. Further, getting the sign of a ratnum is not intended to *apply* the unary function that the sign represents. The basic issue is that *signed integers* are very often implemented as part of a language definition; thus the programmer will not necessarily know how the operating system is treating the signs for integers. Of course, the application problem may also never require negative rationals. At a sufficiently low level of data description, abstraction is not possible due to language and operating system implementation decisions.

The make-function should include all type checking invariants. In the example below, we force an explicit identification of sign (no default permitted), and check that the numerator and denominator are positive integers.

```
make-ratnum[s,n,d] =def=
  if not[member[s,{+, -}]
    then return error-signal[s,"improper-ratnum-sign"]
    else if (not[is-integer[n]] or is-negative[n])
      then return error-signal[n,"improper-ratnum-numerator"]
      else if (not[is-integer[d]] or is-negative[d] or d=0)
        then return error-signal[d,"improper-ratnum-denominator"]
        else return List[s,n,d]
```

Should we later elect to change the implementation of ratnums, the above four functions are all that need to be changed. Say we decide to use 3 element arrays:

```
make-ratnum[s,n,d] =def=
  let a = Make-array[3] of integers
      a[0] := s
      a[1] := n
      a[2] := d
    return a

get-sign[r] =def=
  return r[0]

get-numer[r] =def=
  return r[1]

get-denom[r] =def=
  return r[2]
```

Should we later decide to include the sign of the rational number as a signed numerator, we still would change only these functions.

```
make-ratnum[n,d] =def=
  let a = Make-array[2] of integers
      a[0] := n
      a[1] := d
    return a

get-sign[r] =def=
  return sign-of[r[0]]

get-numer[r] =def=
  return r[0]

get-denom[r] =def=
  return r[1]
```

Finally, in languages which provide first-class functions, we can define the generic get-functions for an ADS. An example:

```
get-numer =def=
  function[parameters[r], body[r[0]]]
```

Challenge problem: The above model assumes that an integer can be of any precision. We know, however, that computational integers have an associated bit-length (e.g. 32bit, 64bit). Modify the above conceptualization and implementation to include specific implementation constraints such as binary precision, storage and bandwidth precision, and binary encoding.