National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2021/2022

Mission 11a
Generic Arithmetic: Rational Numbers

Release date: 28 March 2022
Due: 28 April 2022, 23:59

Required Files

- generic_arith.py
- mission11a-template.py

Information

Remember Ben and Alyssa's packages introduced during the lecture? Now it's our turn to build a generic arithmetic package!

In this mission, we will investigate how generic operators can help us to manage a system where there are multiple related data types that sometimes interact. We will study and modify a generic number system that works with multiple number data types.

We start by introducing the tagging system that will be used to differentiate types of numbers and then list the arithmetical operations we would like each of our number packages to support. Next, we examine how these operations are implemented for one of the packages and why they should be implemented the way they are. With that basis, we study how rational/complex numbers maybe be represented reliably and finally we try our hand at extending the package.

This mission will provide you with an opportunity to work together with another classmate. There are two variants for this mission: Mission 11a and Mission 11b, which are mostly equivalent. **You are only required to complete and submit one of the two**, and your friend should do and submit the other. Do not submit your friend's code as your own submission in the part not done by you.

You will share your code with your partner for Mission 12, when it is unlocked. Please state clearly your partner for Mission 11 in your current submission and also in Mission 12, if you choose to pair up with another student. You could also choose to do both parts of Mission 11 by yourself. However, you will not be able to earn twice the EXP. **If you do both missions, you will only be awarded the EXP for the earlier submission.**

This mission consists of **six** tasks.

Overview of Number Types and Tagging

Missions 11 and 12 will make heavy use of data-directed programming. For the forgetful, data-directed programming involves examining the tag of the data and finding the correct operation from a table. We will be using the table `_operation_table` for this.

Note: The _ prefix for `_operation_table` means by convention, for internal use. You should not be writing code that modifies the `_operation_table` directly.

The types of data we will be dealing with are as follows:

Generic-Num This is the umbrella term for any type of number, and is either one of the three types of numbers below.

Generic-Ord, Generic-Rat, Generic-Com These are abstractions of an ordinary number[1], a rational number, and a complex number, respectively. These are the three types of numbers we will be dealing with in this mission. Each of these types contains a **tag** (`ordinary`, `rational`, or `complex`) as well as the **representation** for their type of number, as illustrated below [2]:

- Generic-Ord = ($\{$`ordinary`$\}$ $\times$ RepOrd)
- Generic-Rat = ($\{$`rational`$\}$ $\times$ RepRat)
- Generic-Com = ($\{$`complex`$\}$ $\times$ RepCom)

Rep-Ord, Rep-Rat, Rep-Com These are the representations of an ordinary number, a rational number, and a complex number, respectively. The actual implementation is up to the programmer. In these missions, for example, Rep-Ord is actually just a Python Number. This will be described in greater clarity later.

The commands `put` and `get` are available to update `_operation_table` when needed. You do not need to be concerned about how `put` and `get` are implemented.

Set Up and Installation of Packages

The source code for this mission can be found in the file `generic_arith.py`. There is a large amount of code for you to manage; you will need to learn how to master code organization well enough to know what you need to understand and what you don't.

The generic arithmetic system is organized into groups of related definitions labeled as "packages". A package generally consists of all the functions for handling a particular type of data, or for handling the interface between packages. These packages are enclosed in package installation functions that install internally defined functions into the table `_operation_table`. This ensures there will be no conflict if a function with the same name is used in another package, allowing packages to be developed separately with minimal coordination of naming conventions.

You will need to edit some portions of the packages to add functionality to the system. Be aware that in a few places, which will be explicitly noted below, we will modify (for the better!) the organization of the generic arithmetic system described in the text.

[1] Note the distinction between "generic ordinary numbers" and "generic numbers". A generic ordinary number is a generic number, but not vice versa.

[2] $A \times B$ refers to the Cartesian product of A and B, and $\{x\}$ is a set containing x. You may read "{t} $\times$ RepX" as "RepX pre-fixed with a tag t".

Each of the three number packages you will be dealing with in this and the next mission must be *installed* before you can use any functions defined within them. For example, to install the ordinary number package, you will need to evaluate the expression `install_ordinary_package()`. This needs to be done anytime you alter the `install_ordinary_package` function definition. Installation of the other packages is done in a similar manner.

Generic Arithmetical Operations

In order to create a generic number, we must use one of the following functions: `create_ordinary`, `create_rational` or `create_complex`.

Some familiar arithmetic operations on generic numbers:

```
def add(x, y):
    return apply_generic("add", x, y)
def sub(x, y):
    return apply_generic("sub", x, y)
def mul(x, y):
    return apply_generic("mul", x, y)
def div(x, y):
    return apply_generic("div", x, y)
```

These functions are of type `(Generic-Num, Generic-Num) -> Generic-Num`. That's because these functions take in two `Generic-Num` types and return a `Generic-Num`. `apply_generic` is simply a function that finds the correct operation based on the type of `x` and `y`, strips `x` and `y` of their type tags and *applies* the operation on their contents. It is so named as it is a function that can be used on any generic number.

We also have

```
# negate: (Generic-Num) -> Generic-Num
def negate(x):
    return apply_generic("negate", x)

# is_zero: (Generic-Num) -> Boolean
def is_zero(x):
    return apply_generic("is_zero", x)

# is_equal: (Generic-Num, Generic-Num) -> Boolean
def is_equal(x, y):
    return apply_generic("is_equal", x, y)
```

In this way, the user can simply call any of the above-mentioned operations without needing to check for the type of their arguments, as the relevant function will be looked up in `_operation_table`.

Generic Ordinary Number Package

To handle ordinary numbers, we must first decide how they are to be represented. Since Python already has an elaborate system for handling numbers, the most straightforward

thing to do is to use it by letting the representation for numbers be the underlying Python representation. That is, let RepOrd be any Python integer or float. This allows us to define the methods that handle generic ordinary numbers simply by calling the Python operators `+`, `-`, ...

```
def install_ordinary_package():
    def make_ord(x):
        return tag(x)
    def tag(x):
        return attach_tag("ordinary", x)

    # add, sub, mul, div: (RepOrd, RepOrd) -> Generic-Ord
    # Note the different output type. It is because the output
    # is tagged before returning.
    def add_ord(x, y):
        return make_ord(x + y)
    def sub_ord(x, y):
        return make_ord(x - y)
    def mul_ord(x, y):
        return make_ord(x * y)
    def div_ord(x, y):
        return make_ord(x / y)
    ...
```

See `generic_arith.py` for the full listing.

The internally defined binary functions `add_ord`, `sub_ord`, `mul_ord` and `div_ord` that manipulate pairs of generic ordinary numbers are of type

$$(\text{RepOrd}, \text{RepOrd}) \to (\{\texttt{ordinary}\} \times \text{RepOrd}) = \text{Generic-Ord}$$

The technique of data-directed programming guarantees that all internal functions will only be called when asked to handle a Generic-Ord. This additionally guarantees that an internal function will only receive data of type RepOrd since `apply_generic` strips the tags for us.

Illustration of Operation Table Lookup

This is one way of visualising the `_operation_table`:

	"ordinary"	("ordinary")	("ordinary", "ordinary")	("rational", "rational")	...
"make"	make_ord	-	-	-	...
"add"	-	-	add_ord	add_rat	...
"mul"	-	-	mul_ord	mul_rat	...
"negate"	-	negate_ord	-	-	...

Suppose the user wants to add two Generic-Ords, `ord1` and `ord2`, each of which has a `"ordinary"` tag attached to it:

```
>>> add(create_ordinary(5), create_ordinary(10))
```

`apply_generic` then finds the function `add_ord` in the table after looking for the entry in `"add"` and `("ordinary", "ordinary")`. The execution of `add_ord` uses Python's native

operator '+' on the contents of `ord1` and `ord2`, which are Python numbers since their tags have been stripped. It then makes a Generic-Ord from the answer by re-attaching an `"ordinary"` tag to it, and returns it.

Task 1: Compound Generic Operations (3 marks)

With these operations, compound generic operations can be defined, such as:

```
def square(x):
    return mul(x, x)
```

1. What are the types of the input and output of the generic `square` operation?
2. Why would we prefer to define `square` in the above way, rather than:

   ```
   def square(x):
       return apply_generic("square", x)
   ```

Hint: Think about the work involved for the programmer for each definition, especially for the different number packages.

Task 2: Constructor vs. Generic Number Operators (3 marks)

In the ordinary number package, a generic number operator is indexed by the name of the operator and a tuple of strings. For example, the add operator is indexed by `'add_ord'` and `('ordinary', 'ordinary')`; negation is indexed by `'negate_ord'` and `('ordinary', )`.

In contrast, the constructor that creates an ordinary number is indexed by `'make_ord'` and just a string `'ordinary'`. Explain why we have such a difference.

Hint: Consider the differences in the process of the creation of a Generic-Num, such as `create_ordinary`, and the operations we can apply on Generic-Num, such as `add`. How is `make_ord` invoked, and how is `add_ord` invoked?

Generic Rational Number Package

To build the generic rational number package, we begin by specifying the representation of rational numbers as *pairs* of Generic-Num's:

$$\text{RepRat} = \text{Generic-Num} \times \text{Generic-Num}$$

A rational number can be created with constructor `create_rational` of type $(\text{Generic-Num}, \text{Generic-Num}) \rightarrow \text{Generic-Rat}$. Similar to generic ordinary number, a generic rational number is marked by tag `rational` as:

$$\text{Generic-Rat} = \{\texttt{rational}\} \times \text{RepRat}$$

Note: You may have noticed at this point that it is possible to create complicated constructs that do not necessarily make sense using our current framework. For example,

one could create a generic rational number with generic complex numbers as components, or worse, create generic complex numbers with generic complex numbers as components. So, in the interest of simplicity and mathematical clarity, we will restrict ourselves with the following intuitive rules for the rest of the problem set:

- A generic rational number will always be formed only with integer generic ordinary numbers as numerator and denominator.
- A generic complex number will be formed only with generic ordinary numbers or generic rational numbers for their real and imaginary parts.

Task 3: Creation of generic rational numbers (3 marks)

There's a right way and a wrong way to create a generic rational number. Here are two tries at producing 9/10. Which is the right way?

```
first_try = create_rational(9, 10)
second_try = create_rational(create_ordinary(9), create_ordinary(10))
```

What happens when you use the wrong way to produce 9/10 and 3/10 and then try to add them? Why does this happen?

Task 4: Structure of generic rational numbers (4 marks)

Produce expressions that define `r2_7` to be the generic rational number whose numerator part is 2 and whose denominator part is 7, and `r3_1` to be the generic rational number whose numerator is 3 and whose denominator is 1. (2 marks)

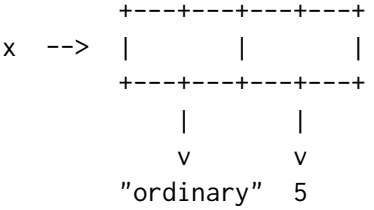
Assume that the expression

```
csq = square(sub(r2_7, r3_1))
```

is evaluated. Draw a box and pointer diagram that represents `csq`. (2 marks)

As an example, the following is a box and pointer diagram that represents `x`, a Generic-Ord number:

```
x = create_ordinary(5)
```



You are free to upload image files for this question.

Task 5: Naming (2 marks)

Within the generic rational number package, the internal `add_rat` function handled the addition operation. Why is it not possible to name this function `add`?

Task 6: Upgrading the package (9 marks)

Modify function `install_rational_package` with

1. a function `negate_rat` suitable for installation as a method for generic `negate` for generic rational numbers (1 mark)
2. a function `is_zero_rat` suitable for installation as a method for generic `is_zero` for generic rational numbers (1 mark)
3. a function `is_eq_rat` suitable for installation as a method for generic `is_equal` for generic rational numbers (1 mark)

Include the types of the input and output of each of these functions as comments accompanying your definition. (1 mark per function)

Extend the generic rational number package to handle `negate`, `is_zero` and `is_equal` using the above functions. (1 mark per extension)

Test that `is_equal` works properly on generic rational numbers. In particular verify the following:

```
>>> is_equal(sub(r1_2, mul(r2_4, r1_2)), add(r1_8, r1_8))
True
```