CS220 Programming Principles

Homework No. 4

Due: May 10, 2017 9:00AM

(18 questions x 3 points = 54 points)

There is a larger amount of code for you to manage in this problem set, and the code makes heavy use of data-directed techniques. We do not intend for you to study it all - and you may run out of time if you try. This problem set will give you an opportunity to acquire a key professional skill: mastering the code organization well enough to know what you need to understand and what you don't need to understand.

- The generic arithmetic system is organized into groups of related definitions labeled as "packages." A package generally consists of all the procedures for handling a particular type of data, or for handling the interface between packages. These packages are enclosed in package installation procedures that install internally defined procedures in the table `operation-table`. This ensures there will be no conflict if a procedure with the same name is used in another package, allowing packages to be developed separately with minimal coordination of naming conventions.
- The complete code is found at the KLMS page. Load the code into DrRacket. **Set language mode to R5RS.** You will need to edit some portions of this code to add functionality to the system.
- Be aware that in a few places, which will be explicitly noted below, this problem set modifies (for the better!) the organization of the generic arithmetic system described in the text.

When you write up this problem set, include in your report answers to the following questions:

- What was the total time you spent working this problem set?
- Indicate the source and nature of any other help you may have received in the problem solutions, including students not in the class and any portions of the subject archives you may have referenced.
- Also include in your report implementations of the packages you worked on, examples/test cases showing that the required functionality to your system was successfully implemented and comments/descriptions of your approaches or code as necessary. Make sure that your write-up is neat, clear and concise so that your TA can follow and understand your work. Simply appending a transcript to your code files is not a write-up.

The Basic Generic Arithmetic System

Generic Numbers

There are three kinds, or subtypes, of generic numbers: generic ordinary numbers, generic rational numbers, generic complex numbers and generic polynomials. Elements of these subtypes consist of a tag (**`number`**, **`rational`**, or **`complex`**) and a data structure representing the element. For

example, a generic ordinary number consists of a tag number and its contents, RepNum, which corresponds to the actual Scheme representation for a generic ordinary number when implemented.

We represent:

- A generic ordinary number as: Generic-OrdNum =({**number**} x RepNum)
- a generic rational number as: Generic-Rational =({**rational**} x RepRat)
- and a generic complex number as: Generic-Complex =({**complex**} x RepCom)

Generic Numbers

A generic number (Generic-Num)[1] is either a generic ordinary number (Generic-OrdNum), a generic rational number (Generic-Rational) or a generic complex number (Generic-Complex), as expressed in a type equation:

Generic-Num = ({**number**} x RepNum) U ({**rational**} x RepRat) U ({**complex**} x RepCom).

The type tagging mechanism is the simple one described on p. 175 of the SICP book, and the apply-generic is the one without coercions described in section 2.4.3. The code for these is in **hw04skel_2017.scm**.

We will also assume that the commands **put** and **get** are available to automagically update the table of methods around which the system is designed. You need not be concerned in this problem set how put and get are implemented[2].

Generic Arithmetic

Some familiar arithmetic operations on generic numbers are:

```
; add, sub, mul, div: (Generic-Num, Generic-Num) → Generic-Num.
(define (add x y) (apply-generic 'add x y))
(define (sub x y) (apply-generic 'sub x y))
(define (mul x y) (apply-generic 'mul x y))
(define (div x y) (apply-generic 'div x y))
```

We also have

```
; negate: Generic-Num → Generic-Num.
(define (negate x) (apply-generic 'negate x))
; =zero?: Generic-Num → Sch-Bool.
(define (=zero? x) (apply-generic '=zero x))
; equ?: (Generic-Num, Generic-Num) → Sch-Bool.
(define (equ? x y) (apply-generic 'equ? x y))
```

---

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

2 This will be explained when we come to section 3.3.3 of the SICP book.

The generic predicate **`=zero?`** tests whether a generic number is equal to zero, and the generic predicate **`equ?`** tests whether two generic number operands are equal.

Exercise 1 With these operations, compound generic operations can be defined, such as

```
(define (square x) (mul x x))
```

(a) What is the type of the generic square operation?
(b) Why is square not defined as

```
(define (square x) (apply-generic 'square x))
```

For the remainder of this problem set, you will be dealing with three types of packages. Each must be *installed* before you can use any procedures defined within them. For example, to install the ordinary number package, you will need to evaluate the expression: **`(install-number-package).`** This needs to be done anytime you alter the **`install-number-package`** procedure definition. Installing the other packages is done in a similar manner.

Ordinary Number Package

To handle ordinary numbers, we must first decide how they are to be represented. Since Scheme 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 Scheme representation. Namely, let

RepNum = Sch-Num:

This allows us to define the methods that handle generic ordinary numbers simply by calling the Scheme primitives +, -, . . . , as in section 2.5.1. So we can immediately define interface procedures between RepNum's and the Generic Number System:

```
;;; the ordinary number package
define (install-number-package)
  (define (tag x) (attach-tag 'number x))
  (define (make-number x) (tag x))
  (define (negate x) (tag (- x)))
  (define (zero? x) (= x 0))
  (define (add x y) (tag (+ x y)))
  (define (sub x y) (tag (- x y)))
  (define (mul x y) (tag (* x y)))
  (define (div x y) (tag (/ x y)))
  (put 'make 'number make-number)
  (put 'negate '(number) negate)
  (put '=zero? '(number) zero?)
  (put 'add '(number number) add)
  (put 'sub '(number number) sub)
```

```
(put 'mul '(number number) mul)
(put 'div '(number number) div)
'done)
```

The internally defined binary procedures **add**, **sub**, **mul** and **div** that manipulate pairs of ordinary numbers are of type (RepNum, RepNum) → ({**number**} x RepNum) = Generic-OrdNum.

Exercise 2 What are the types of the **make-number**, **negate**, **zero?** procedures that are defined internally in the **install-number-package** procedure?

Exercise 3 The procedures for number operations (and others) are keyed by lists of pairs of symbols, e.g., (**number number**), but the **make-number** operation is keyed by just the symbol **number**. How does the tag-type system manage to handle this difference? Note that the procedure for the generic **negate** operation is keyed by a list consisting of a single symbol.

To install the ordinary number methods in the generic operations table, we evaluate

```
(install-number-package)
```

The ordinary number package should provide a means for a user to create generic ordinary numbers, so we include a user-interface procedure[3] of type Sch-Num → ({**number**} x RepNum), namely,

```
(define (create-number x) ((get 'make 'number) x))
```

Implementing equ? for Ordinary Numbers

Your goal is to implement the necessary code so that the generic number operation equ? works successfully for arguments of type Generic-OrdNum.

Exercise 4A Modify the procedure **install-number-package** to include the definition of a procedure called **=number?** which will be suitable for installation as a method for **equ?** to use to handle generic ordinary numbers. Include the type of **=number?** as a comment line accompanying your definition.

Exercise 4B After extending the ordinary number package with **equ?** and installing this package, test that it works properly on generic ordinary numbers. In particular, verify that if we define

```
(define n2 (create-number 2))
(define n4 (create-number 4))
(define n6 (create-number 6))
```

---

[3] In Exercise 2.78 in the SICP book, the implementation of the type tagging system is modified to maintain the illusion that generic ordinary numbers have a **number** tag, without actually attaching the tag to Scheme numbers. This implementation has the advantage that generic ordinary numbers are represented exactly by Scheme numbers, so there is no need to provide the user-interface procedure **create-number**. Note that in Section 2.5.2, the SICP book implicitly assumes that this revised implementation of tags has been installed. In this problem set we stick to the straightforward implementation with actual **number** tags.

then the expression **`(equ? n4 (sub n6 n2))`** is true.

Rational Number Package

You will be working with a Rational Number package that is similar to the one described in section 2.1.1. One difference is that the problem set rational number package uses generic arithmetic operations rather than the primitive ones (+, -, etc.) so that numerators and denominators can be arbitrary generic numbers and are not limited to only integers (ordinary numbers). The situation is like that in Section 2.5.3 in which the use of generic operations in **`add-terms`** and **`mul-terms`** allowed manipulation of polynomials with arbitrary coefficients.
We begin by specifying the representation of rationals as *pairs* of Generic-Num's:

RepRat = Generic-Num x Generic-Num

with constructor **`make-rat`** of type (Generic-Num, Generic-Num) → RepRat. Note that **`make-rat`** does not attempt to reduce rationals to lowest terms as in Section 2.1 because we are allowing arbitrary numerators and denominators, and **`gcd`** only makes sense when numerator and denominator are integers.

The basic arithmetic procedures within the Rational Number Package, **`add-rat`**, **`sub-rat`**, etc., are of type (RepRat, RepRat) → RepRat. Note the distinction between the internal procedures such as **`add-rat`** which create untagged objects of type RepRat and procedures such as **`add-rational`** that create tagged generic rationals.

These and other procedures that comprise the Rational Number Package are defined within the procedure **`install-rational-package`**:

```
;;; the rational number package
(define (install-rational-package)
        (define (make-rat n d) (cons n d))
        (define (numer x) (car x))
        (define (denom x) (cdr x))
        (define (add-rat x y)
          (make-rat (add (mul (numer x) (denom y))
                         (mul (denom x) (numer y)))
                    (mul (denom x) (denom y))))
        (define (sub-rat x y)
          (make-rat (sub (mul (numer x) (denom y))
                         (mul (denom x) (numer y)))
                    (mul (denom x) (denom y))))
        (define (mul-rat x y)
          (make-rat (mul (numer x) (numer y))
                    (mul (denom x) (denom y))))
        (define (div-rat x y)
```

```
    (make-rat (mul (numer x) (denom y))
              (mul (denom x) (numer y))))
(define (tag x) (attach-tag 'rational x))
(define (make-rational n d) (tag (make-rat n d)))
(define (add-rational x y) (tag (add-rat x y)))
(define (sub-rational x y) (tag (sub-rat x y)))
(define (mul-rational x y) (tag (mul-rat x y)))
(define (div-rational x y) (tag (div-rat x y)))
(put 'make 'rational make-rational)
(put 'add '(rational rational) add-rational)
(put 'sub '(rational rational) sub-rational)
(put 'mul '(rational rational) mul-rational)
(put 'div '(rational rational) div-rational)
'done)
```

To install the rational methods in the generic operations table, we evaluate:

```
(install-rational-package)
```

Notation: Let **`rn/d`** be the rational whose numerator is **`n`** and whose denominator is **`d`**.
The external procedure **`create-rational`** (of type RepRat = (Generic-Num, Generic-Num) → ({**`rational`**} x RepRat) = Generic-Rational) allows users to create generic rational numbers:

```
(define (create-rational n d) ((get 'make 'rational) n d))
```

Exercise 5A There's a right way and a wrong way to create a rational number. Here are two tries at producing **`r9/10`**. Which is the right way?

```
(define first-try (create-rational 9 10))
(define second-try (create-rational (create-number 9) (create-number 10)))
```

What happens when you use the wrong way to produce **`r9/10`** and **`r3/10`** and then try to add them? Why does this happen?

Exercise 5B Produce expressions that define **`r2/7`** to be the rational number whose numerator is 2 and whose denominator is 7, and **`r3/1`** to be the rational number whose numerator is 3 and whose denominator is 1. Remember that the arguments to **`create-rational`** are of type **Generic-Num**. Assume that the expression

```
(define rsq (square (sub r2/7 r3/1)))
```

is evaluated. Draw a box and pointer diagram that represents rsq.

Exercise 6 Within the Ordinary Number Package, the internal add procedure handled the addition operation. The corresponding procedure in the Rational Number Package is **`add-rational`**. Why was it not possible to similarly give this procedure the name **`add`**?

Exercise 7A Modify procedure **`install-rational-package`** with:

a procedure **`negate-rat`** suitable for installation as a method for generic **`negate`** for rationals,
a procedure **`=zero-rat?`** suitable for installation as a method for generic **`=zero?`** for rationals,
a procedure **`=rational?`** suitable for installation as a method for generic **`equ?`** for rationals,

Include the type signatures of each of these procedures as comments accompanying your definition. Include a copy of your modified **`install-rational-package`** procedure with your solutions.

Exercise 7B Install **`equ?`** as an operator on rationals in the generic arithmetic package. Test that it works properly on general rational numbers. In particular verify that **`(equ? (sub r1/1 (mul r1/2 r1/3)) (add r1/2 r1/3))`** is true.

Operations Across Different Types of Numbers

At this point all the methods installed in our system require all operands to have the subtype - all number, or all rational. There are no methods installed for operations combining operands with distinct subtypes. For example,

```
(define n3 (create-number 3))
(equ? n3 r3/1)
```

will return a "no method" error message because there is no equality method at the subtypes (**`number rational`**). We have not built into the system any connection between the number 3 and the rational 3/1.

Some operations across distinct subtypes are straightforward. For example, to combine a rational with a number, *n*, coerce *n* into the rational *n*/1 and combine them as rationals.

Exercise 8 Within the Rational Number Package define a procedure

`repnum->reprat` : RepNum → RepRat

that coerces the ordinary number *n* into a rational number whose numerator is the ordinary number *n* and whose denominator is the ordinary number 1.

Procedure **`RRmethod->NRmethod`** makes it possible to obtain a (RepNum, RepRat) → *T* method from a (RepRat, RepRat) → *T* method, for ***any*** type *T* :

```
(define (RRmethod->NRmethod method)
```

```
(lambda (num rat)
  (method
    (repnum->reprat num)
    rat)))
```

Exercise 9 Define the corresponding procedure **`RRmethod->RNmethod`** that for any type *T* can be used to obtain a (RepRat, RepNum) → *T* method from a (RepRat, RepRat) → *T* method.

Exercise 10A Using **`RRmethod->NRmethod`** and **`RRmethod->RNmethod`**, modify the Rational Number Package to define methods for generic **`add`**, **`sub`**, **`mul`**, and **`div`** at argument types (**`number rational`**) and at argument types (**`rational number`**). Also define **`equ?`** for these argument types.

Exercise 10B Install your new methods. Test them on **`(equ? n3 r3)`** and

```
(equ? (sub (add n3 r2/7) r2/7) n3)
```

Complex Number Package - for Your Information Only

You will be working with a Complex Number package (rectangular form only) that is similar to the one described in section 2.5.1. One difference is that the problem set complex number package uses generic arithmetic operations rather than the primitive ones (+, -, etc.) so that the real and imaginary parts can be arbitrary generic numbers and are not limited to only integers (ordinary numbers).

We begin by specifying the representation of complex numbers as *pairs* of Generic-Num's:

RepCom = Generic-Num x Generic-Num

with constructor **`make-com`** of type (Generic-Num, Generic-Num) → RepCom.
The basic arithmetic procedures within the Complex Number Package, **`add-com`**, **`sub-com`**, etc., are of type (RepCom, RepCom) → RepCom. Note the distinction between the internal procedures such as **`add-com`** which create untagged objects of type RepCom and procedures such as **`add-complex`** that create tagged generic complexs.

These and other procedures that comprise the Complex Number Package are defined within the procedure **`install-complex-package`**:

```
;; complex number package in rectangular form (a+bi)
(define (install-complex-package)
  (define (make-com r i) (cons r i))
  (define (real x) (car x))
  (define (imag x) (cdr x))
  (define (add-com x y)
    (make-com (add (real x) (real y))
```

```
            (add (imag x) (imag y))))
  (define (sub-com x y)
    (make-com (sub (real x) (real y))
              (sub (imag x) (imag y))))
  (define (mul-com x y)
    (make-com (sub (mul (real x) (real y))
                   (mul (imag x) (imag y)))
              (add (mul (real x) (imag y))
                   (mul (real y) (imag x)))))
  (define (div-com x y)
    (let ((com-conj (complex-conjugate y)))
      (let ((x-times-com-conj (mul-com x com-conj))
            (y-times-com-conj (mul-com y com-conj)))
        (make-com (div (real x-times-com-conj) (real y-times-com-conj))
                  (div (imag x-times-com-conj) (real y-times-com-conj))))))
  (define (complex-conjugate x)
    (make-com (real x) (negate (imag x))))
  (define (tag x) (attach-tag 'complex x))
  (define (make-complex n d) (tag (make-com n d)))
  (define (add-complex x y) (tag (add-com x y)))
  (define (sub-complex x y) (tag (sub-com x y)))
  (define (mul-complex x y) (tag (mul-com x y)))
  (define (div-complex x y) (tag (div-com x y)))
  (put 'make 'complex make-complex)
  (put 'add '(complex complex) add-complex)
  (put 'sub '(complex complex) sub-complex)
  (put 'mul '(complex complex) mul-complex)
  (put 'div '(complex complex) div-complex)
  'done)
```

To install the complex methods in the generic operations table, we evaluate:

```
(install-complex-package)
```

Notation: Let **`ca+bi`** be the complex whose real part is **`a`** and whose imaginary part is **`b`**.
The external procedure **`create-complex`** (of type RepCom = (Generic-Num, Generic-Num) → ({**`complex`**} x RepCom) = Generic-Complex) allows users to create generic complex numbers:

```
(define (create-complex r i) ((get 'make 'complex) r i))
```

Interfacing Packages

In this section, you will look at getting complex numbers and rational numbers to co-exist together. Just like each package was extended so that methods can deal with arguments of different types, you

will now consider how to allow rational numbers to have complex numerators and denominators, and complex numbers to have rational real and imaginary parts.

Exercise 11 Consider the mathematical operation of dividing **`1+3i`** by 5. How would you express this using the complex number package? What do you expect will happen?
How would you express this using the rational number package? What do you expect will happen?
Try out your expressions, what happens? Is this what you predicted?

Complex Number Division

Exercise 12 What about the mathematical operation of dividing **`1+3i`** by **`1+2i`**? What do you think will happen with each package? If you compute **`(div c1+3i c1+2i)`** with the complex number package defined in the problem set code, you get something like:

```
;Value: (complex (number . 1.4) number . .2)
```

Why does this happen? What element of which package gets called that causes this?

Exercise 13 In some cases, we might want the complex number to actually have rational real and imaginary parts. There may be several ways to handle this. Choose one solution you discussed, implement it and note its advantages/disadvantages. Test it on **`(div c1+3i c1+2i)`**.

Equality of Complex and Rational Numbers

Exercise 14 Design and implement a means for testing equality between complex numbers and rational numbers. What methods need to be added to your package? Do any methods have to be added to the table? Test to see if **`(3+2i)/5 ?=? (3/5)+(2/5)i`**.