MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996-97

Problem Set 6

A Generic Arithmetic Package

Issued: 8 October 1996

Written solutions due: in recitation on Friday, October 25, 1996

Tutorial preparation for: week of October 21, 1995

Reading: Section 2.4 and 2.5 of the notes, plus attached code.

This problem set is based on Sections 2.4 and 2.5 of the notes, which discuss a generic arithmetic system that is capable of dealing with rational functions (quotients of polynomials). You should study and understand these sections and also carefully read and think about this handout before attempting to solve the assigned problems.

The generic arithmetic system is organized into a number of pieces. The complete code is attached at the end of the handout. All of this code will be loaded into Scheme when you load the files for this problem set. You will need to edit some portions of this code to add functionality to the system.

There is a larger amount of code for you to manage in this problem set than in previous ones. Furthermore, 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.

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.

Use e-mail to send your tutor your PreLab work, computer listings of all the procedures you write in lab, and transcripts showing that the required functionality was added to your system. The transcript should include enough tests to demonstrate the functionality of your modifications and show that they work properly.

Please keep track of the total time you spend working this problem set and the total time you spend in lab, and turn in these times along with your PreLab and Lab work.

Generic Arithmetic

The basic generic arithmetic system

There are three kinds, or *subtypes*, of generic numbers in the system of this Problem Set: generic ordinary numbers, generic rational numbers, and generic polynomials. Elements of these subtypes are tagged items with one of the tags `number`, `rational`, or `polynomial`, followed by a data structure representing an element of the corresponding subtype. For example, a generic ordinary number has tag `number` and another part, called its *contents*, which represents an ordinary number.

We represent:

- a generic ordinary number as: Generic-OrdNum = ({`number`} × RepNum)
- a generic rational number as: Generic-Rational = ({`rational`} × RepRat)
- and a generic polynomial as: Generic-Polynomial = ({`polynomial`} × RepPoly).

A generic number (Generic-Num) is either a generic ordinary number (Generic-OrdNum), a generic rational number (Generic-Rational), or a generic polynomial (Generic-Polynomial), as expressed in a type equation:

$$\text{Generic-Num} = (\{\texttt{number}\} \times \text{RepNum}) \cup (\{\texttt{rational}\} \times \text{RepRat}) \cup (\{\texttt{polynomial}\} \times \text{RepPoly}).$$

The type tagging mechanism is the simple one described on p. 166 of the text, and the `apply-generic` is the one *without coercions* described in section 2.4.3. The code for these is in `types.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 needn't be concerned in this problem set how `put` and `get` are implemented[1].

Some familiar arithmetic operations on generic numbers are

```
(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))
```

The type of an arithmetic operation is a specification of the types of its operands and the type of its value. The four operations above all take a pair of operands of type Generic-Num and return a value of type Generic-Num and are thus all of type (Generic-Num, Generic-Num) → Generic-Num. We also have

```
(define (negate x) (apply-generic 'negate x))
(define (=zero? x) (apply-generic '=zero x))
(define (equ? x y) (apply-generic 'equ? x y))
```

The `negate` operation takes a generic number operand and returns a generic number result, and is thus of type Generic-Num → Generic-Num. The generic predicate `=zero?` tests whether a generic

[1] This will be explained when we come to section 3.3.3 of the Notes.

number is equal to zero and is hence of type Generic-Num → Sch-Bool, where the type of Sch-Bool is one of the two Scheme Boolean values `true` and `false`. The generic predicate `equ?` tests whether two generic number operands are equal and is hence of type (Generic-Num, Generic-Num) → Sch-Bool.

Using these operations, compound generic operations can be defined, such as

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

Tutorial Exercise 6.1A What is the type of the generic `square` operation?

Tutorial Exercise 6.1B Why is `square` not defined as

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

Tutorial Exercise 6.1C The procedures for the generic operations `add, sub,` etc. are keyed by lists of pairs of symbols, e.g., `(number number)`, but the `make` 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.

Packages

The code for the generic number system of this problem set has been organized in `ps6-code.scm` 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.

Ordinary numbers

To install 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, namely, let

$$\text{RepNum} = \text{Sch-Num}.$$

i.e., let the representation for numbers be the underlying Scheme representation. 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 (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 tag)
  (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 $(\text{RepNum}, RN) \to (\{\texttt{number}\} \times \text{RepNum}) = \text{Generic-OrdNum}$, where `number` is the tag attached to the Scheme representation of a number in forming a generic ordinary number.

Tutorial Exercise 6.1D What are the types of the `make-number, negate, zero?` procedures that are defined internally in the `install-number-package` procedure?

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[2] of type $\text{Sch-Num} \to (\{\texttt{number}\} \times \text{RepNum})$, namely,

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

PreLab exercise 6.2A To test the arithmetic equality of its arguments, the generic arithmetic package includes the equality predicate `equ?` of type

$$\texttt{equ?} : (\text{Generic-Num}, \text{Generic-Num}) \to \text{Sch-Bool}.$$

Modify the procedure `install-number-package` to include a definition of an `=number?` procedure suitable for installation as a method for `equ?` to handle generic ordinary numbers. Include the type of `=number?` in comments accompanying your definition.

[2] In Exercise 2.78 in the text, 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 following Exercise 2.77, the text 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.

Lab exercise 6.2B Install `equ?` as an operator on numbers in the generic arithmetic 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))
```

then the expression

```
(equ? n4 (sub n6 n2))
```

is true.

Rational number package

The second piece of the system is a Rational Number package like the one described in section 2.1.1. However in the package included in the Problem Set, generic arithmetic operations, rather than the primitive primitive `+`, `-`, etc., are used to combine numerators and denominators. This difference is important, because it allows "rationals" whose numerators and denominators are arbitrary generic numbers, rather than only integers or 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

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

with constructor of type $(\text{Generic-Num}, \text{Generic-Num}) \rightarrow \text{RepRat}$. The procedures that comprise the Rational Number Package are defined within the procedure `install-rational-package` (see below).

Note that the internally defined `make-rat` procedure does not attempt to reduce rationals to lowest terms as in Section 2.1 because `gcd` makes sense only in certain cases – such as when numerator and denominator are integers – but we are allowing arbitrary numerators and denominators.

The basic arithmetic procedures within the Rational Number Package, `add-rat`, `Sub-rat`, etc., are of type $(\text{RepRat}, \text{RepRat}) \rightarrow \text{RepRat}$.

The rational Package also provides a means for a user to create Generic Rationals by attaching the tag `rational` to a RepRat object. Specifically the external procedure `create-rational` (of type $\text{RepRat} = (\text{Generic-Num}, \text{Generic-Num}) \rightarrow (\{\texttt{rational}\} \times \text{RepRat}) = \text{Generic-Rational}$):

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

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

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

where:

```
;;; 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)
```

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.

PreLab Exercise 6.3A Produce expressions that define `r2/7` to be the rational number whose numerator is 2 and whose denominator is 7, and `r3` to be the rational number whose numerator is 3 and whose denominator is 1. Assume that the expression

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

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

Tutorial Exercise 6.3B Within the Ordinary Number Package, the internal `add` procedure handled the addition operation. The corresponding procedure in the Rational Number Package is `add-rational`. Since there are not name conflicts between the internally defined procedures, why was it not possible to give this procedure the name `add`?

PreLab Exercise 6.4A Modify procedure `install-rational-package` to handle the unary operations `negate-rat` of type RepRat $\rightarrow$ ({`rational`} $\times$ RepRat) = Generic-Rational that negates a rational and `=zero-rat?` of type RepRat $\rightarrow$ Sch-Bool that tests whether a rational number is equal to zero. Also include a definition of an `=rational?` procedure suitable for installation as a method allowing generic `equ?` to handle generic rational numbers. Include the type of `=rational?` in comments accompanying your definition. Include a copy of your modified `install-rational-package` procedure with your solutions.

Lab Exercise 6.4B 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 if

- `r1/2` is the rational whose numerator is 1 and whose denominator is 2,
- `r1/3` is the rational whose numerator is 1 and whose denominator is 3,

then the expression

```
(equ? (sub r1 (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)
```

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.

PreLab exercise 6.5A Within the Rational Number Package define a procedure

`repnum->reprat` : RepNum $\rightarrow$ 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) $\rightarrow T$ method from a (RepRat, RepRat) $\rightarrow T$ method, for *any* type T:

```
(define (RRmethod->NRmethod method)
  (lambda (num rat)
    (method
     (repnum->reprat num)
     rat)))
```

PreLab Exercise 6.5B Define the corresponding procedure `RRmethod->RNmethod` that for any type T can be used to obtain a $(\text{RepRat}, \text{RepNum}) \to T$ method from a $(\text{RepRat}, \text{RepRat}) \to T$ method.

PreLab Exercise 6.5C Using `RRmethod->NRmethod`, modify the Rational Number Package to define methods for generic `add`, `sub`, `mul`, and `div` at argument types (`number rational`). Define methods for these operations at argument types (`rational number`). Also define `equ?` for these argument types.

Lab exercise 6.5D Install your new methods. Test them on `(equ? n3 r3)` and

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

Polynomials

The procedure defined within the procedure `install-polynomial-package` (see below) comprise the main part of the Polynomial Package. Since the number of procedures required to manipulate polynomials is relatively large, some additional procedures are defined "top level" in this problem set. Within the generic arithmetic system a polynomial is an object with the tag `polynomial`.

Within the Polynomial Package, a polynomial is specified by its variable and its "termlist". The representation of a polynomial satisfies the following type equations:

$$\text{RepPoly} = \text{Variable} \times \text{RepTerms}$$

$$\text{RepTerms} = \text{Empty-Term-List} \cup (\text{RepTerm} \times \text{RepTerms})$$

$$\text{RepTerm} = \text{Scheme-NatNum} \times \text{Generic-Num}$$

where

- Variable is a representation of the variable of the polynomial
- RepTerm is a representation of the terms of the polynomial
- Empty-Term-List corresponds to the empty term list
- (RepTerm × RepTerms) indicates a term joined to a term list
- Scheme-NatNum is a non-negative integer corresponding to the exponent of a term
- Generic-Num is a generic number corresponding to the coefficient of the term.

Thus a termlist is either empty or consists of one or more terms, where each term includes a specification of the order of the term and the coefficient of the term.

Although "termlists" are currently represented using the format preferred for `sparse` polynomials as described in Section 2.5.3, the code treats them as abstract data structures, with their own constructors and selectors, so that a different representation could be used if required.

```
(define (install-polynomial-package)
  (define (tag poly) (attach-tag 'polynomial poly))
  (define (make-polynomial var terms)
    (tag (make-poly var terms)))
  (define (variable? x) (symbol? x))
  (define (same-variable? v1 v2)
    (and (variable? v1) (variable? v2) (eq? v1 v2)))
  (define (add-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (add-termlists (term-list p1)
                                  (term-list p2)))
        (error "Polys not in same var -- ADD-POLY"
               (list p1 p2))))
  (define (mul-poly p1 p2)
    (if (same-variable? (variable p1) (variable p2))
        (make-poly (variable p1)
                   (mul-termlists (term-list p1)
                                  (term-list p2)))
        (error "Polys not in same var -- MUL-POLY"
               (list p1 p2))))
  (define (add-polynomial p1 p2) (tag (add-poly p1 p2)))
  (define (mul-polynomial p1 p2) (tag (mul-poly p1 p2)))
  (put 'make 'polynomial make-polynomial)
  (put 'add '(polynomial polynomial) add-polynomial)
  (put 'mul '(polynomial polynomial) mul-polynomial)
  'done)
```

```
;;; addition of termlists
(define (add-termlists L1 L2)
  (cond ((empty-termlist? L1) L2)
        ((empty-termlist? L2) L1)
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term
                   t1 (add-termlists (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term
                   t2 (add-termlists L1 (rest-terms L2))))
                 (else
                  (adjoin-term
                   (make-term (order t1)
                              (add (coeff t1) (coeff t2)))
                   (add-termlists (rest-terms L1)
                                  (rest-terms L2)))))))))
;;; multiplication of termlists
(define (mul-termlists L1 L2)
  (if (or (empty-termlist? L1) (empty-termlist? L2))
      (make-empty-termlist)
      (add-termlists (mul-term-by-all-terms (first-term L1) L2)
                     (mul-termlists (rest-terms L1) L2))))

;;; create a polynomial
(define (create-polynomial var terms)
  ((get 'make 'polynomial) var terms))
```

To implement term lists as data structures we have the constructors

$$\texttt{make-empty-termlist} : \text{Empty-type} \to \text{RepTerms}$$
$$\texttt{adjoin-term} : (\text{RepTerm}, \text{RepTerms}) \to \text{RepTerms},$$

and selectors

$$\texttt{first-term} : \text{RepTerms} \to \text{RepTerm}$$
$$\texttt{rest-terms} : \text{RepTerms} \to \text{RepTerms}$$

Note that `(make-empty-termlist)` returns a representation of an "empty"[3] termlist.

In this problem set, we modify the definition of `mul-term-by-all-terms` given on P. 206 of the notes. The new definition is

[3] The Empty-type has no elements. The type statement

$$\texttt{make-an-element} : \text{Empty-type} \to T$$

indicates that the procedure **`make-an-element`** takes no arguments, and evaluating **`(make-an-element)`** returns a value of type T. Such procedures are sometimes called "thunks". There wasn't any special need to use a thunk as constructor for empty term lists – a constant equal to the empty term list would have served as well (perhaps better?) – but it serves as a reminder that term lists are created differently than Scheme's lists.

```
(define (mul-term-by-all-terms t1 tlist)
  (map-terms
     (lambda (term) (mul-terms t1 term))
     tlist))
```

where

```
(define (mul-terms t1 t2)
  (make-term
    (+ (order t1) (order t2))
    (mul (coeff t1) (coeff t2))))
```

PreLab Exercise 6.6A Procedure `map-terms` applies a procedure to each term in a termlist. Define it.

Tutorial Exercise 6.6B What is the `+` procedure used to combine the orders of the term arguments to `mul-terms` rather than the generic `add` procedure?

PreLab exercise 6.6C Define a procedure `create-numerical-polynomial` which, given a variable name, `x`, and list of ordinary numbers, returns a generic polynomial in `x` with the coefficients specified by the scheme numbers in the list. To create the numerical polynomial $7x^2 + 3$ you would evaluate the expression

```
(create-numerical-polynomial 'x '(7 0 3))
```

Lab exercise 6.6D Evaluate your definition of `map-terms`, thereby completing the definition of multiplication of generic polynomials.

Use `create-numerical-polynomial` to define `p1` to be the generic polynomial

$$p_1(x) = x^3 + 5x^2 - 2.$$

Use the generic `square` operator to compute the square of `p1`, and the square of its square. Turn in the the **pretty-printed** results of the squarings, as computed in lab.

PreLab Exercise 6.7A Add procedures to the Polynomial Number Package that can be used with the generic operators `negate, =zero?,` and `equ?`. Also add procedures so that the `sub` operation can handle polynomials.[4]

[4]There is also no method for polynomial `div`, but this is more problematical since polynomials are not closed under division, e.g., dividing $x + 1$ by x^2 yields a rational function

$$\frac{x+1}{x^2}$$

which is not equivalent to any polynomial.

Lab Exercise 6.7B Test your generic `negate`, `sub`, and `equ?` procedures on the expressions `(negate p1)`, `(add p1 (negate p1))`, and `(equ? (square (sub p1 p1)) (sub p1 p1))`.

There are still no methods installed which work with operands of mixed types. This means that generic arithmetic on polynomials with generic coefficients of different types is likely to fail. Consider the polynomial $p_2(z,x) = p_1(x)z^2 + 5z + 3$ which is defined in `ps6-ans.scm` as:

```
(define p2
  (create-polynomial
   'z
   (adjoin-term
      (make-term 2 p1)
      (adjoin-term
         (make-term 1 (create-number 5))
         (adjoin-term
            (make-term 0 (create-number 3))
            (make-empty-termlist))))))
```

Now squaring `p2` will generate a "no method" error message, because there is no method for multiplying the numerical coefficients 5 and 3 by the polynomial coefficient `p1`. A method that would work in our system would be to represent $p_2(z,x)$ as the equivalent polynomial $p_2(z,x) = p_1(x)z^2 + q(x)z + r(x)$ where $q(x)$ and $r(x)$ are the *constant* polynomials in x, $q(x) = 5$ and $r(x) = 3$.

PreLab Exercise 6.8A Redefine p2 so that it can be squared successfully.

Lab exercise 6.8B Use the generic `square` operator to compute the square of `p2` and the square of the square of `p2`. Turn in the **pretty-printed** results of the squarings, as computed in lab.

Polynomial Evaluation

Polynomials are generic numbers on the one hand, but on the other hand, they also describe *functions* which can be applied to generic numbers. For example, the polynomial $p_1(x) = x^3 + 5x^2 - 2$ evaluates to 26 when $x = 2$.

The procedure `apply-polynomial` can be defined as follows

```
(define (apply-polynomial p generic-number)
  (apply-terms
    (term-list (contents p))
    generic-number))
```

where, recognizing that a polynomial is implicitly constructed from its terms by addition, we define

```
(define (apply-terms termlist generic-number)
  (if (empty-termlist? termlist)
      ...
      (add ... ...)))

(define (apply-term term generic-number)
  (mul (coeff term)
       (power generic-number (order term))))

(define (power n k)
  (if (< k 1)
      (create-number 1)
      (mul n (power n (- k 1)))))
```

PreLab Exercise 6.9A Complete the definition of `apply-terms`.

Lab Exercise 6.9B Test your definition by applying p_1 to -1 and $1/2$.

Lab exercise 6.9C If the polynomial $x + 1$ is substituted for z in p_2, the result should be a polynomial in x. What result do you obtain when you apply p_2 to $x + 1$? How can you rectify the situation?

The section on "Feedback" in the *Don't Panic* manual describes how to send Email from the 6.001 lab. Send your answer buffer to your tutor. Do this by starting a mail message and using the Edwin `M-x insert-buffer` command. Don't forget to include a reply email address (or else put your name in the message) so that your tutor will know that the message is coming from you.

Collaboration and Pedagogy

Indicate the names of your collaborators, if any, on this assignment. Indicate the source and nature of any other help you may have received in the problem solutions, including students not in 6.001 and any portions of the subject archives you may have referenced. (Remember, we encourage *open* collaboration among 6.001 students.)