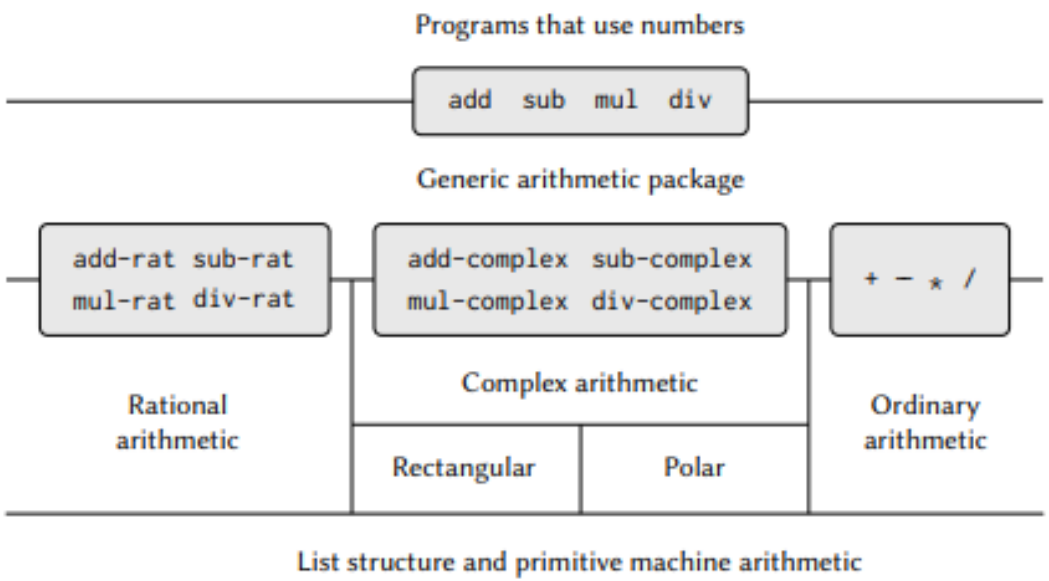
Till now, we saw how to design systems in which data objects can be represented in more than one way.

Extending that idea, we shall see how one can not only define operations
that are generic over different representations of same type
but also to
that are generic over different kinds of arguments (different types).



We will now use data-directed techniques to construct a package of arithmetic operations that incorporates all the arithmetic (ordinary,rational and complex) packages we have already constructed.

We would like, for instance, to have a generic addition procedure add that acts like ordinary primitive addition + on ordinary numbers, like add-rat on rational numbers, and like add-complex on complex numbers.

```
;The generic arithmetic procedures are defined as follows:
(define (add x y) (apply-generic 'add x y))
```

For brevity, we shall focus on add procedure. The rest of the arithmetic operations are dealt analogously.

First, we install a package for handling ordinary numbers. We will tag these with the symbol *scheme-number*.

Since these operations each take two arguments, they are installed in the table keyed by the list (scheme-number scheme-number)

.

```
(define (install-scheme-number-package)
  (define (tag x) (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
   .
   .
   .
  (put 'make 'scheme-number (lambda (x) (tag x)))
  'done)

; Users of the Scheme-number package will create
;(tagged) ordinary numbers by means of the procedure:

(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
```

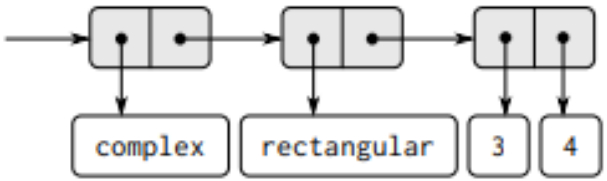
```
(define (install-rational-package)
  ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
      (cons (/ n g) (/ d g))))
  (define (add-rat x y)
    (make-rat (+ (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))
  .
  .
  .

;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  .
  .
  .

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

In case of ordinary numbers, the arithmetic operations are primitive arithmetic procedures (so there is no need to define extra procedures to handle the untagged numbers), whereas in case of rationals and

In case of complex numbers, we would have two tags.



The outer tag (complex) is used to direct the number to the complex package. Once within the complex package, the next tag (rectangular) is used to direct the number to the rectangular package.

Name-Conflicts:- In the above packages, we used add-rat, add-complex, and the other arithmetic procedures exactly as originally written. Once these definitions are internal to different installation procedures, however, they no longer need names that are distinct from each other: we could simply name them add, sub, mul, and div in both packages.

The operations we have defined so far treat the different data types as being completely independent.
The operations that cross the type boundaries, such as the addition of a complex number to an ordinary number, also need to be taken care of.

The operations we have defined so far treat the different data types as being completely independent.
The operations that cross the type boundaries, such as the addition of a complex number to an ordinary number, also need to be taken care of.

Brute-Force

One way to handle cross-type operations is to design a different procedure for each possible combination of types for which the operation is valid.
For example, we could extend the complex-number package so that it provides a procedure for adding complex numbers to ordinary numbers and installs this in the table using the tag (complex scheme-number)

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
  (make-from-real-imag (+ (real-part z) x) (imag-part z)))
(put 'add '(complex scheme-number)
     (lambda (z x) (tag (add-complex-to-schemenum z x))))
```

- The cost of introducing a new type is not just the construction of the package of procedures for that type but also the construction and installation of the procedures that implement the cross-type operations.
  Thus, this method undermines our ability to combine separate packages additively.
- Formulating coherent policies on the division of responsibility (as to which package will handle which cross-type operation) can also be an overwhelming task in the presence of multiple packages/cross-type operations.

Although, this works as a worst-case scenario option, when other cleaner approaches fail.

Often the different data types are not completely independent, and there may be ways by which objects of one type may be viewed as being of another type.
This process is called **coercion**.

For example, if we are asked to add an ordinary number to a complex number, we can view the ordinary number as a complex number whose imaginary part is zero. Thus, we can transform the problem to that of combining two complex numbers.

$$6 + (-3 + 3i) = (6 + 0i) + (-3 + 3i) = (3 + 3i)$$

In general, we can implement this idea by designing coercion procedures that transform an object of one type into an equivalent object of another type.

```
(define (scheme-number->complex n)
  (make-complex-from-real-imag (contents n) 0))
```

We install these coercion procedures in a special coercion table, indexed under the names of the two types:

```
(put-coercion 'scheme-number
              'complex
              scheme-number->complex)
```

Some of the slots in the coercion table may be empty. For eg. ~~complex $\longrightarrow$ scheme-number~~.
Once the coercion table has been set up, we can handle coercion in a uniform manner by modifying the apply-generic procedure.

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    ; looks in the operation-type table
    ; and applies the resulting procedure if one is present
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          ; For simplicity, assume two argument
          ; cross-type operations are encountered
          (if (= (length args) 2)
              ; Extracting type tags and contents
              ; of the two arguments
              (let ((type1 (car type-tags))
                    (type2 (cadr type-tags))
                    (a1 (car args))
                    (a2 (cadr args)))
                ; Check if one of the arg
                ; can be co-erced to another type
                ; if possible and then apply procedure
                (let ((t1->t2 (get-coercion type1 type2))
                      (t2->t1 (get-coercion type2 type1)))
                  (cond (t1->t2
                         (apply-generic op (t1->t2 a1) a2))
                        (t2->t1
                         (apply-generic op a1 (t2->t1 a2)))
                        (else (error "No method for these types"
                                     (list op type-tags))))))
              (error "No method for these types"
                     (list op type-tags))))))
```

Modified *apply-generic* checking for the possibility of coercion.

Advantage:-

1. No need to define explicit cross-type operations.
2. Need to write only one procedure per pair of types rather than a different procedure for each collection.

   However, one might be required to write several coercion procedures to relate the types (possibly $\binom{n}{2}$ procedures for a system with n types). In practice, we can usually get by with fewer than $\binom{n}{2}$ coercion procedures.

There may be situations for which our coercion scheme is not general enough.
Even when neither of the objects to be combined can be converted to the type of the other it may still be possible to perform the operation by converting both objects to a third type. (For example 'triangle' and 'square' would be converted to a third type 'polygon')

The coercion scheme presented above relied on the existence of natural relations between pairs of types.
Often there is more "global" structure in how the different types relate to each other.
For example, $\mathbb{N} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

The hierarchy mentioned above is of a very simple kind, in which each type has at most one supertype and at most one subtype.
Such a structure is called a *tower*.

In tower structure, if there are n types we need $n - 1$ coercion procedures between types adjacent in the tower.

For example, if we want to add an integer to a complex number, we need not explicitly define a special coercion procedure integer→complex.

Instead, we define

integer→rational number,
rational→real, and
real→complex.

The system then transforms the integer into a complex number through these steps and then add the two complex numbers.

We can redesign our apply-generic procedure in the following way: For each type, we need to supply a raise procedure, which "raises" objects of that type one level up in the tower.

Thus, when the system is required to operate on objects of different types it can successively raise the lower types until all the objects are at the same level in the tower.

Another advantage of a tower is that we can easily implement the notion that every type "inherits" all operations defined on a supertype.

For instance, if we do not supply a special procedure for finding the real part of an integer, we should nevertheless expect that real-part will be defined for integers by virtue of the fact that integers are a subtype of complex numbers.

If the required operation is not directly defined for the type of the object given, we raise the object to its supertype and try again.

Another advantage of a tower over a more general hierarchy is that it gives us a simple way to "lower" a data object to the simplest representation.
For example $6 + 0i$, should be lowered from complex number to integer.

If the hierarchies form a partial order instead of total order(tower), it makes implementation of coercion procedures less straight-forward.

Multiple-supertypes means that there is no unique way to “raise” a type in the hierarchy.

Since there generally are multiple subtypes for a type, there is a similar problem in coercing a value “down” the type hierarchy.

We will represent polynomials using a data structure called a *poly*, which consists of a variable and a collection of terms.
$5x^3 + 3x + 7$ would be represented as
$(x\ (5\ 0\ 3\ 7))$ or
$(x\ ((5\ 3)\ (3\ 1)\ (0\ 7)))$

We assume that we have selectors variable and term-list that extract those parts from a poly and a constructor make-poly that assembles a poly from a given variable and a term list. A variable will be just a symbol, so we can use the same-variable? procedure to compare variables.

```
(define (add-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1)
                 (add-terms (term-list p1) (term-list p2)))
      (error "Polys not in same var: ADD-POLY" (list p1 p2))))
```

To incorporate polynomials into our generic arithmetic system, we need to supply them with type tags.
We'll use the tag polynomial, and install appropriate operations on tagged polynomials in the operation table.

Polynomial addition is performed termwise. Terms of the same order (i.e., with the same power of the indeterminate) must be combined.

```
(adjoin-term
 (make-term (order t1)
            (add (coeff t1) (coeff t2)))
 (add-terms (rest-terms L1)
            (rest-terms L2)))))))))
```

The most important point to note here is that we used the generic addition procedure add to add together the coefficients of the terms

Notice that, since we operate on terms using the generic procedure add, our polynomial package is automatically able to handle any type of coefficient that is known about by the generic arithmetic package.

If we include a coercion mechanism , then we also are automatically able to handle operations on polynomials of different coefficient types. For example

$$[3x^2 + (2 + 3i)x + 7] \cdot \left[x^4 + \frac{2}{3}x^2 + (5 + 3i)\right].$$

This system is also automatically able to handle polynomial operations such as

$$\left[(y + 1)x^2 + (y^2 + 1)x + (y - 1)\right] \cdot \left[(y - 2)x + (y^3 + 7)\right].$$

The result is a kind of **"data-directed recursion"** in which, for example, a call to mul-poly will result in recursive calls to mul-poly in order to multiply the coefficients.

If the coefficients of the coefficients were themselves polynomials, the data direction would ensure that the system would follow through another level of recursive calls, and so on through as many levels as the structure of the data dictates.

Polynomial algebra is a system for which the data types cannot be naturally arranged in a tower.

For instance, it is possible to have polynomials in x whose coefficients are polynomials in y.

It is also possible to have polynomials in y whose coefficients are polynomials in x. Conversion between the two is possible, however neither of these types is "above" the other in any natural way.

Much of the complexity of such systems is concerned with relationships among diverse types.

It is fair to say that we do not yet completely understand coercion.

In fact, we do not yet completely understand the concept of a data type.

Nevertheless, what we know provides us with powerful structuring and modularity principles to support the design of large systems.