CS339: Abstractions and Paradigms for Programming

Tagging and Message Passing

Manas Thakur
CSE, IIT Bombay

Autumn 2025

Recall the problem we discussed in the last class

- We had two representations for complex numbers (rectangular and polar).
- Each representation seemed more natural for certain operations (rectangular for addition/subtraction, and polar for multiplication/division).
- But name-conflict issues in our scheme forced only one to be used at a time.
- Can we use both the representations together?

Tagging

- Same procedure names for different representations.
  - Easy solution: **Rename** the procedures; suffix '`-rectangular`' with rectangular procedures and '`-polar`' with polar procedures.
- A bigger problem is that both rectangular and polar complex numbers may be roaming around in an indistinguishable manner.
- Let us introduce a mechanism to distinguish rectangular and polar representations of complex numbers.
- **Tag** data items:
  - Rectangular data with '`rectangular`
  - Polar data with '`polar`

Attaching tags

- How do we add a tag to a complex number?
- cons the tag with the existing pair!

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum) (car datum))

(define (contents datum) (cdr datum))
```

- How could we find if a given complex number representation is rectangular or polar?

```
(define (rectangular? z) (eq? (type-tag z) ‘rectangular))

(define (polar? z) (eq? (type-tag z) ‘polar))
```

Revised rectangular implementation

- Tags attached in the constructors:

```
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

- Name conflicts resolved by renaming:

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))
```

Similarly, the revised polar representation

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

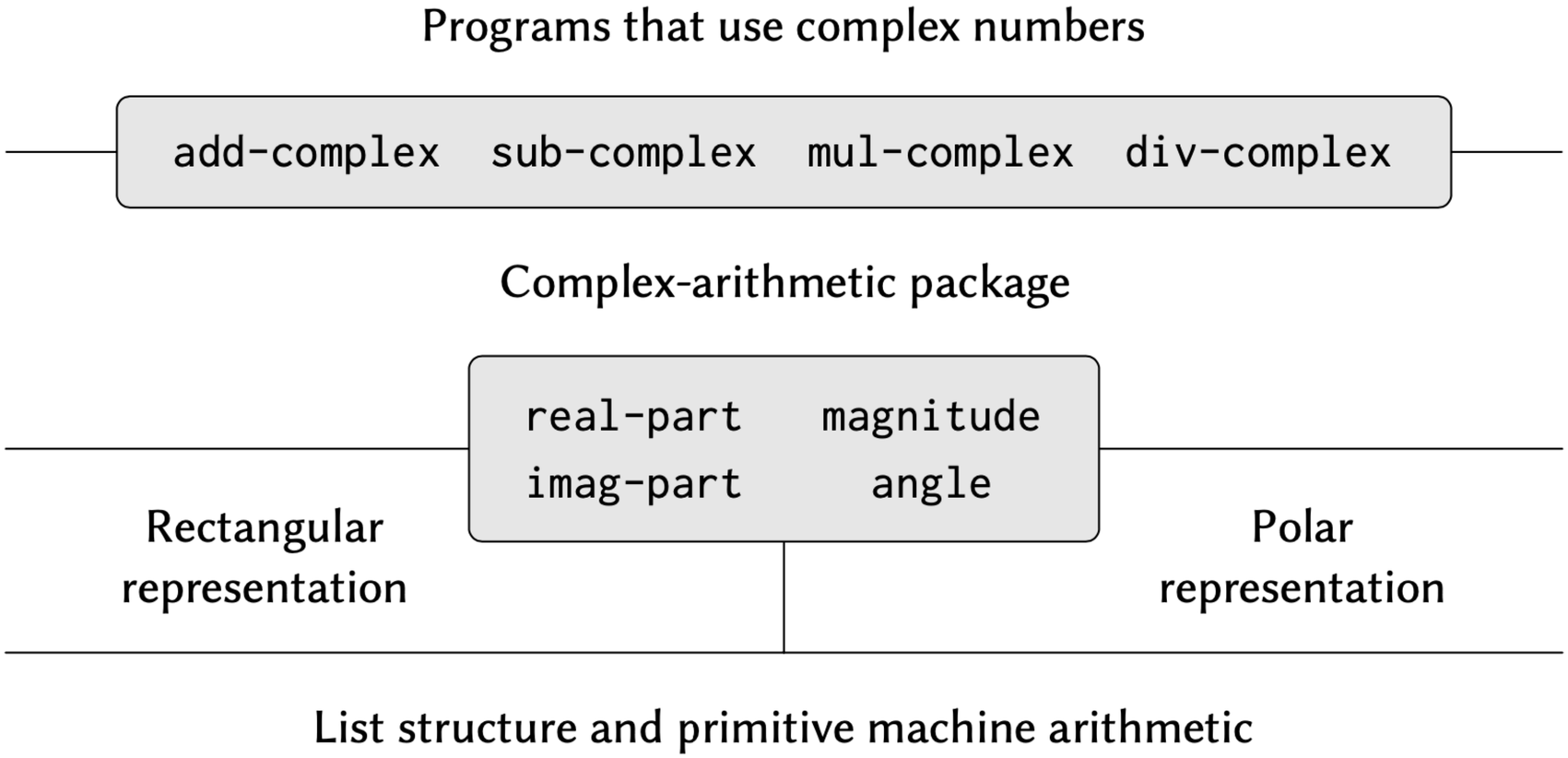
Tagging solves the problem

- Now we can decide which procedure to call using tags:

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z))))
```

- Basically, we are dispatching a procedure based on the type-tag associated with the data object.
- Similar changes can be made for other conflicting procedures.
- Both representations can co-exist peacefully.

Abstractions in our complex number library



Is there anything bad in our scheme?

- What if I want to add 10 more representations?
  - Define new `make-*` procedures and selectors while making sure that the names don't match — umm, painful but obvious.
  - Define a new tag for each representation — looks fine.
  - What about the generic procedures?

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z))))
```

  - Need to add a case for each representation in each generic procedure — ouch!

Message Passing

- Instead of making operations intelligent (read bothersome), make the data objects intelligent:

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op: MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

- What does `make-from-real-imag` return?
- A procedure that takes an argument ‘`op`’ to decide what computation to perform.
- And why is it so interesting?

The picture is not yet over!

Message Passing (Cont.)

- Let us change the generic procedures as follows:

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

  where:

```
(define (apply-generic op arg) (arg op))
```

- Thus, our `make-*` procedure returns another procedure representing an object that *dispatches* the correct procedure based on the message passed to the object (receiver).

Putting it all together

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op: MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

```
(define (apply-generic op arg) (arg op))
```

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
```

```
(define (make-from-mag-ang x y)
  (lambda (op)
    (cond ((eq? op ‘magnitude) x)
          ((eq? op ‘angle) y)
          ((eq? op ‘real-part …
          …)))
```

```
(define n1 (make-from-real-imag 2 3))
(define n2 (make-from-real-imag 3 4))
(define n3 (add-complex n1 n2))
```

The right procedures will be called automagically based on which dispatch procedure lies inside ‘z’!

Back to Object-Oriented Programming

- **Each OOL** provides a mechanism to abstract out objects belonging to a certain kind, such that:
  - the object encapsulates constituent data items as *fields*
  - and the procedures to operate on objects as *methods*

- You create objects and assign values to their fields using constructors, and dispatch methods by passing messages to receivers.

```
(define (make-rat x y)
  (lambda (which)
    (if (= which 0) x y)))

(define (numer n) (n 0))
(define (denom n) (n 1))
(define (mult-rat n1 n2)
  (make-rat (* (numer n1) (numer n2))
            (* (denom n1) (denom n2))))

(define n1 (make-rat 2 3))
(define n2 (make-rat 3 4))
(define n3 (mult-rat n1 n2))
```

Abra-ca-dabra!

```
(define (Rational x y)
  (lambda (msg)
    (cond ((eq? msg 'numer) x)
          ((eq? msg 'denom) y)
          ((eq? msg 'mult-rat)
           (lambda (other)
             (Rational (* x (other 'numer))
                       (* y (other 'denom)))))))))

(define n1 (Rational 2 3))
(define n2 (Rational 3 4))
(define n3 ((n1 'mult-rat) n2))
```

Now ain't they similar?

```
(define (Rational x y)
  (lambda (msg)
    (cond ((eq? msg 'numer) x)
          ((eq? msg 'denom) y)
          ((eq? msg 'mult-rat)
           (lambda (other)
             (Rational (* x (other 'numer))
                       (* y (other 'denom))))))))

(define n1 (Rational 2 3))
(define n2 (Rational 3 4))
(define n3 ((n1 'mult-rat) n2))
```

Notice we got both:

- Packaging
- Dispatch

```
class Rational {
  int x; int y;
  Rational(int x, int y) {
    this.x = x; this.y = y;
  }
  int numer() { return x; }
  int denom() { return y; }
  Rational mult-rat(Rational other) {
    return new Rational(
      this.numer() * other.numer(),
        this.denom() * other.denom());
  }
}
Rational n1 = new Rational(2,3);
Rational n2 = new Rational(3,4);
Rational n3 = n1.mult-rat(n2);
```

- **Preserve** this class to understand the **crux** of some of the fundamentals of the **OO paradigm**.