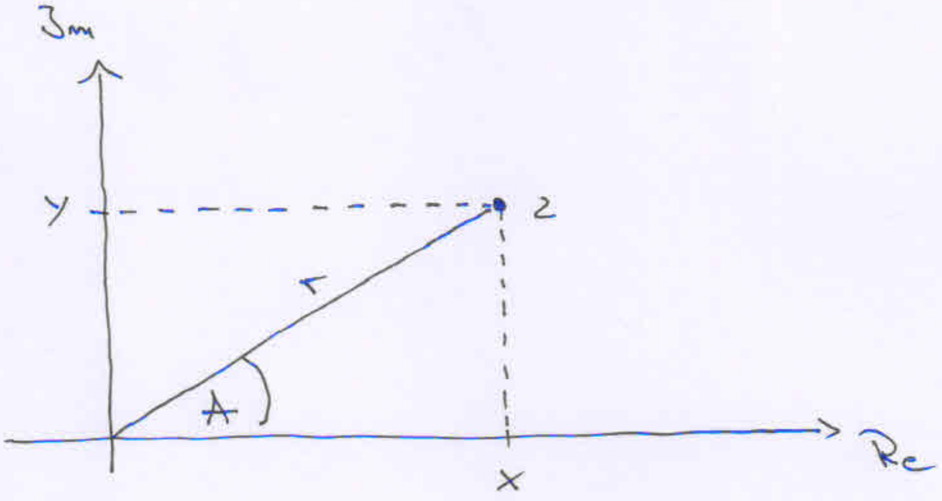
§4 Czym są dane?

Przykład: Liczby zespolone

$z = x + iy$, $i = \sqrt{-1}$ ⟶ z "=" (x, y)

$z_1 \pm z_2 = (x_1 \pm x_2) + i(y_1 \pm y_2)$



$z_1 * z_2 = r_1 \cdot r_2 \cdot e^{i(A_1 + A_2)}$

↳ postać algebraiczna
(część rzeczywista x, część urojona y)
postać wykładnicza
(moduł r, argument A)

$x = r \cdot \cos A$,

$y = r \cdot \sin A$

$r = \sqrt{x^2 + y^2}$

A "=" $\arctan\left(\frac{y}{x}\right)$

```
(define (+c z1 z2)
   (make-rectangular
        (+ (real-part z1) (real-part z2))
        (+ (imag-part z1) (imag-part z2))))

(define (*c z1 z2)
   (make-polar
        (* (magnitude z1) (magnitude z2))
        (+ (angle z1) (angle z2))))
```

(i) Manifest Types

```
        (rectangular . (1 . 5))
        (polar . (7 . 3))
```

→ 3

(ii) Message passing

Liczba zespolona, to funkcja, która "odpowiada" na pewne wiadomości.

→ 4

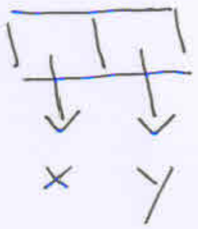
```
;;; Complex Numbers

;;; Arithmetic Operators

(define (+c z1 z2)
  (make-rectangular (+ (real-part z1) (real-part z2))
                    (+ (imag-part z1) (imag-part z2))))

(define (-c z1 z2)
  (make-rectangular (- (real-part z1) (real-part z2))
                    (- (imag-part z1) (imag-part z2))))

(define (*c z1 z2)
  (make-polar (* (magnitude z1) (magnitude z2))
              (+ (angle z1) (angle z2))))

(define (/c z1 z2)
  (make-polar (/ (magnitude z1) (magnitude z2))
              (- (angle z1) (angle z2))))


;;; Adding Type Information to Complex Numbers

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

(define (type datum) (car datum))

(define (contents datum) (cdr datum))

(define (rectangular? z) (eq? (type z) 'rectangular))

(define (polar? z) (eq? (type z) 'polar))

(define (make-rectangular x y)
  (attach-type 'rectangular (cons x y)))

(define (make-polar x y)
  (attach-type 'polar (cons x y)))


;;; Getting the Components

(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))))

(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))))
```

```
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))))

(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))))

(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (car z)) (square (cdr z)))))

(define (angle-rectangular z) (atan (cdr z) (car z)))

(define (real-part-polar z) (* (car z) (cos (cdr z))))

(define (imag-part-polar z) (* (car z) (sin (cdr z))))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))
```

```
;;; Complex Numbers

;;; Arithmetic Operators (same as before)

(define (+c z1 z2)
  (make-rectangular (+ (real-part z1) (real-part z2))
                    (+ (imag-part z1) (imag-part z2))))

(define (-c z1 z2)
  (make-rectangular (- (real-part z1) (real-part z2))
                    (- (imag-part z1) (imag-part z2))))

(define (*c z1 z2)
  (make-polar (* (magnitude z1) (magnitude z2))
              (+ (angle z1) (angle z2))))

(define (/c z1 z2)
  (make-polar (/ (magnitude z1) (magnitude z2))
              (- (angle z1) (angle z2))))


(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else (error "Unknown message in make-rectangular: " m))))
  dispatch)

(define (make-polar x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) (* x (cos y)))
          ((eq? m 'imag-part) (* x (sin y)))
          ((eq? m 'magnitude) x)
          ((eq? m 'angle) (atan y x))
          (else (error "Unknown message in make-polar: " m))))
  dispatch)


(define (real-part obj) (obj 'real-part))
(define (imag-part obj) (obj 'imag-part))
(define (magnitude obj) (obj 'magnitude))
(define (angle obj) (obj 'angle))
```

```
> (define z1 (make-rectangular 1 2))
z1
> (real-part z1)
=> (z1 'real-part)
=> ((make-rectangular 1 2) 'real-part)
=> ((dispatch m) 'real-part)
=> (cond ((eq? 'real-part 'real-part) 1)
         ((eq? 'real-part 'imag-part) 2)
         ((eq? 'real-part 'magnitude)
              (sqrt (+ (square 1)(square 2))))
         ((eq? 'real-part 'angle)
              (atan 2 1))
         (else (error "......")))

=> 1

> (+c z1 (make-polar 7 3))
=> (make-rectangular
      (+ (real-part z1)
         (real-part (make-polar 7 3)))
      (+ (imag-part z1)
         (imag-part (make-polar 7 3))))
=> ...
```

A listy (pary)?



```
(car (cons x y)) = x
(cdr (cons x y)) = y
(cons (car z) (cdr z)) = z
```

Definicja par!

```
(define (new-cons x y)
    (define (dispatch m)
        (cond ((= m 0) x)
              ((= m 1) y)
              (else   (error "...." ))))
    dispatch)

(define (new-car z) (z 0))

(define (new-cdr z) (z 1))
```

```
> (new-car (new-cons x y))
=> ((new-cons x y) 0)
=> ((dispatch m) 0)
=> (cond ((= 0 0) x)
         ((= 0 1) y)
         (else    (error "....")))

=> x
```

↳ równania definicji par są spełnione, czyli mamy realizację par przy pomocy funkcji.

↳ Między funkcjami a danymi nie ma różnicy! (Też liczby można zrealizować przy pomocy funkcji)

```
(define (cons x y) (lambda (m) (m x y)))
(define (car z) (z (lambda (p q) p)))
(define (cdr z) (z (lambda (p q) q)))
```