

Les 8: Generische operatoren

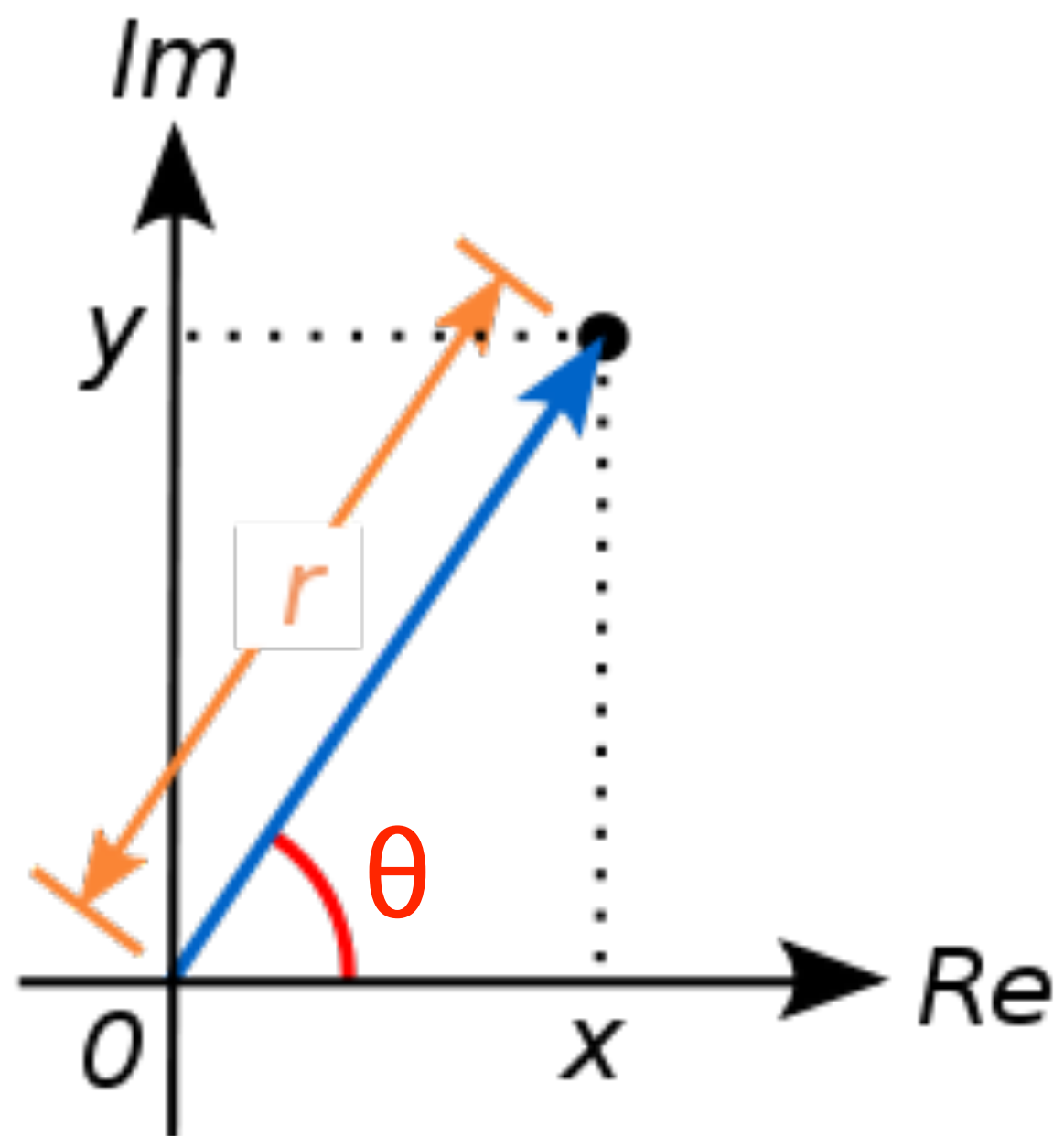
Deel 1

Les 8: Generische operatoren

We bestuderen eerst hoe het mogelijk is om twee verschillende voorstellingen voor hetzelfde data type naast elkaar te laten bestaan. We gebruiken als voorbeeld de complexe getallen waarvoor zowel de Cartesische notatie als de notatie als poolcoördinaten gangbaar zijn. We introduceren daarbij de techniek van manifeste typering en data gericht programmeren.

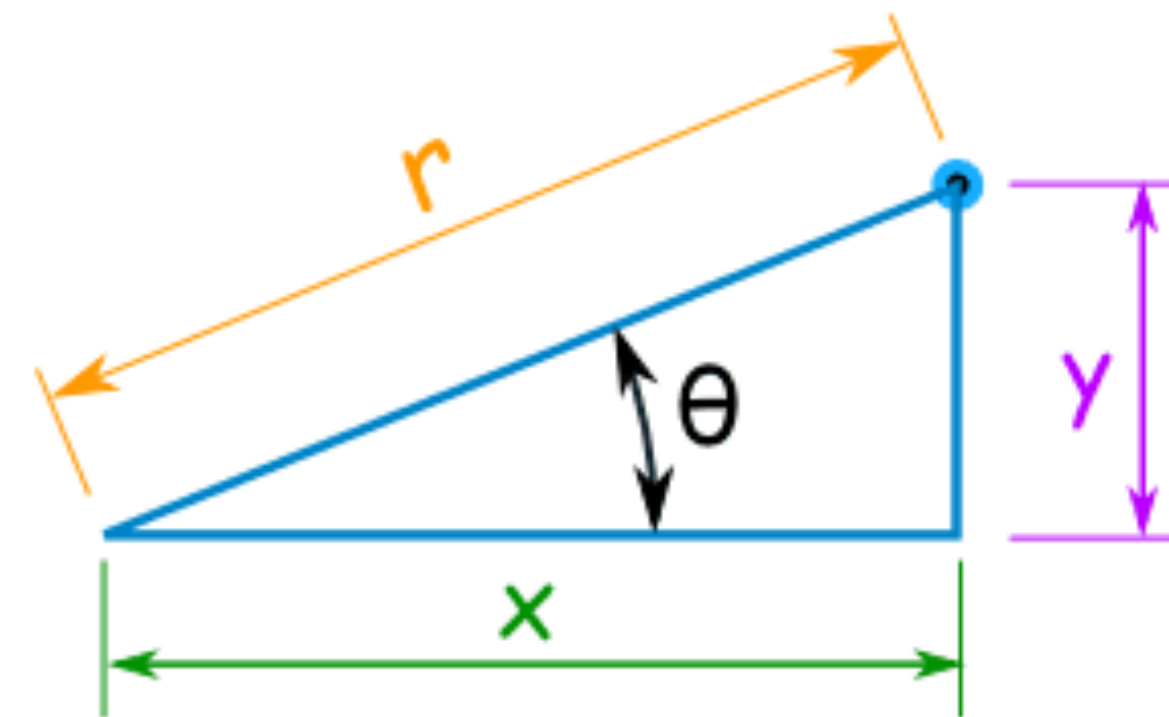
Daarna bestuderen we generische operatoren, d.w.z operatoren die over verschillende data-types kunnen werken. We werken als voorbeeld een pakket uit voor rekenkundige operatoren.

Twée notaties voor complexe getallen



Rectangular: x, y

Polar: r, θ



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

$$\theta = \tan^{-1}(y / x)$$

$$x = r \times \cos(\theta)$$

$$y = r \times \sin(\theta)$$

Complexe getallen: Cartesische representatie

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

```
> (define c (make-from-rect 2 3))
> (real-part c)
2
> (imag-part c)
3
> (magnitude c)
3.605551275463989
> (angle c)
0.982793723247329
```

```
> (make-from-rect 2 3)
(2 . 3)
> (make-from-polar 2 3)
(-1.9799849932008908 . 0.2822400161197344)
```

Complexe getallen: representatie met poolcoördinaten

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

```
> (define c (make-from-polar 2 3))
> (angle c)
3
> (magnitude c)
2
> (real-part c)
-1.9799849932008908
> (imag-part c)
0.2822400161197344
```

```
> (make-from-rect 2 3)
(3.605551275463989 . 0.982793723247329)
> (make-from-polar 2 3)
(2 . 3)
```

Complexe getallen: Operators

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

(define (-c z1 z2)
  (make-from-rect
    (- (real-part z1) (real-part z2))
    (- (imag-part z1) (imag-part z2))))

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

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

Overzicht van het systeem

Data abstraction barriers in the complex number system

Use of complex numbers

$+c$ $-c$ $*c$ $/c =c$

Complex-arithmetic package

make-from-rect	make-from-polar
real-part	imag-part
magnitude	angle

Rectangular package

Polar package

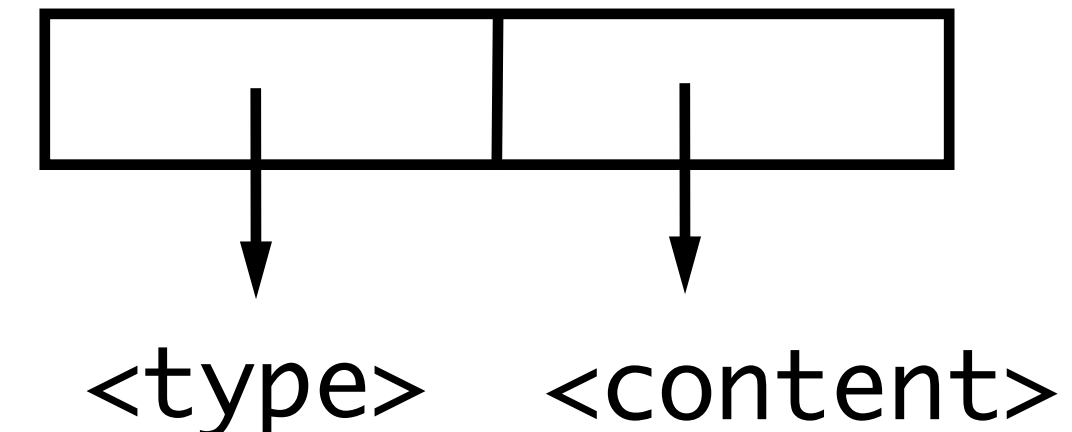
List structure and primitive machine arithmetic

Data labelen – Manifeste typering

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

(define (type object)
  (if (atom? object)
      (error "bad-typed-object -- TYPE" object)
      (car object)))

(define (content object)
  (if (atom? object)
      (error "bad-typed-object -- CONTENT" object)
      (cdr object)))
```



Complexe getallen – Constructors

```
(define (make-from-rect x y)
  (attach-tag 'rectangular (cons x y)))
```

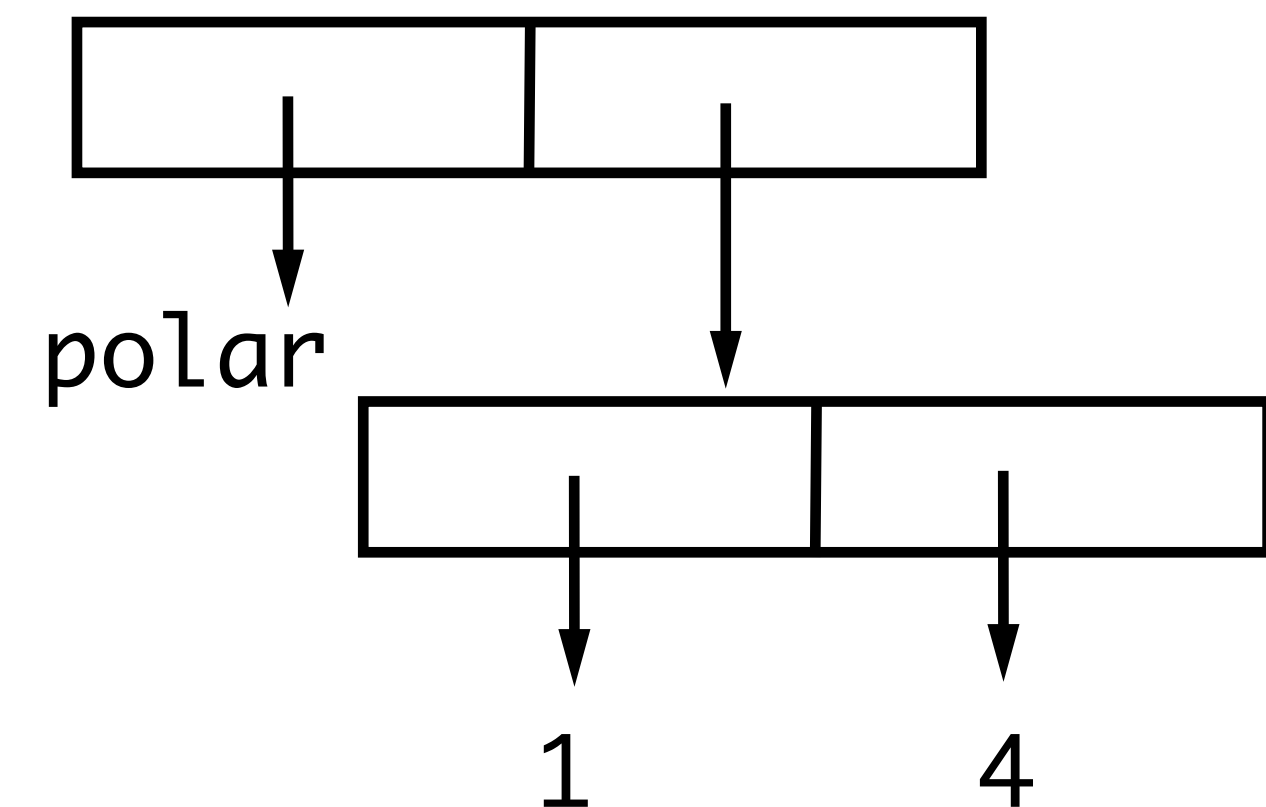
```
(define (make-from-polar r a)
  (attach-tag 'polar (cons r a)))
```

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

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

```
> (make-from-rect 2 5)
(rectangular 2 . 5)
> (make-from-polar 1 4)
(polar 1 . 4)
```

'evidente' keuze voor
de representatie met
het juiste type label



Specifieke selectoren

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

(define (angle-p z)
  (car z))
(define (magnitude-p z)
  (cdr z))
(define (real-part-p z)
  (* (car z) (cos (cdr z))))
(define (imag-part-p z)
  (* (car z) (sin (cdr z))))
```

je moet 'weten' wat
je vast hebt en de
juiste functie zelf
uitzoeken

Generische selectoren

dispatch op basis
van typelabel

```
(define (real-part z)
  (cond
    ((rectangular? z)
     (real-part-r (content z)))
    ((polar? z)
     (real-part-p (content z)))
    (else (error "Unknown type
                  -- REAL-PART" z))))
```

```
(define (imag-part z)
  (cond
    ((rectangular? z)
     (imag-part-r (content z)))
    ((polar? z)
     (imag-part-p (content z)))
    (else (error "Unknown type
                  -- IMAG-PART" z))))
```

```
(define (magnitude z)
  (cond
    ((rectangular? z)
     (magnitude-r (content z)))
    ((polar? z)
     (magnitude-p (content z)))
    (else (error "Unknown type
                  -- MAGNITUDE" z))))
```

```
(define (angle z)
  (cond
    ((rectangular? z)
     (angle-r (content z)))
    ((polar? z)
     (angle-p (content z)))
    (else (error "Unknown type
                  -- ANGLE" z))))
```

Data-gericht Programmeren: idee





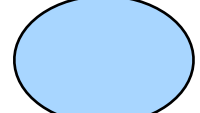




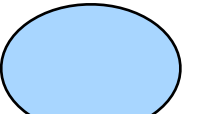

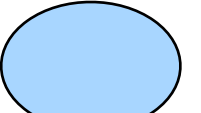
Organiseer tabel operators X types

		Types	
		polar	rectangular
Operators	real-part	real-part-polar	real-part-rectangular
	imag-part	imag-part-polar	imag-part-rectangular
	magnitude	magnitude-polar	magnitude-rectangular
	angle	angle-polar	angle-rectangular

Wanneer operator op bepaald type moet losgelaten worden: haal hem uit tabel en pas toe op argument(en)

Data-gericht Programmeren: uitwerking

Assume existence of : (put <op> <type> <item>)
(get <op> <type>)

	(polar)	(rectangular)	polar	rectangular
real-part				
imag-part				
magnitude				
angle				
make-from-rect				
make-from-polar				

Data-gericht Programmeren:

installatie van beide pakketten

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (make-from-rect x y) (cons x y))
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (angle z) (atan (cdr z) (car z)))
  (define (magnitude z)
    (sqrt (+ (square (car z)) (square (cdr z)))))
  (define (make-from-polar r a)
    (cons (* r (cos a)) (* r (sin a))))
  ;; interfacing
  (define (tag x) (attach-type 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-rect 'rectangular
    (lambda (x y) (tag (make-from-rect x y))))
  (put 'make-from-polar 'rectangular
    (lambda (x y) (tag (make-from-polar x y))))
  'done)
```

```
(define (install-polar-package)
  ;; internal procedures
  (define (make-from-polar r a) (cons r a))
  (define (angle z) (car z))
  (define (magnitude z) (cdr z))
  (define (real-part z) (* (car z) (cos (cdr z))))
  (define (imag-part z) (* (car z) (sin (cdr z))))
  (define (make-from-rect x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interfacing
  (define (tag x) (attach-type 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-rect 'polar
    (lambda (x y) (tag (make-from-rect x y))))
  (put 'make-from-polar 'polar
    (lambda (x y) (tag (make-from-polar x y))))
  'done)
```

Apply

```
> (+ 1 2 3 4 5)  
15  
> (apply + '(1 2 3 4 5))  
15  
> (apply * '(1 2 3 4 5))  
120
```


Define & Lambda list – vast aantal argumenten

```
(define test  
  (lambda (x y z)  
    (+ (* 2 x) (* 2 y) (* 2 z))))
```

OF

```
(define (test x y z)  
  (+ (* 2 x) (* 2 y) (* 2 z)))
```

verwacht exact 3 argumenten

```
> (test 1 2 3)
```

```
12
```

```
> (test 1 2)
```

⊗⊗ *test: arity mismatch;*

the expected number of arguments does not match the given number
expected: 3

given: 2 arguments....:

```
> (test 1 2 3 4 5)
```

⊗⊗ *test: arity mismatch;*

the expected number of arguments does not match the given number
expected: 3

given: 5 arguments....:

Define & Lambda list – variabel aantal argumenten

```
(define test  
  (lambda x  
    (do ((rest x (cdr rest))  
        (result 0 (+ result (* 2 (car rest))))  
        ((null? rest) result))))
```

OF

```
(define (test . x)  
  idem
```

```
> (test 1 2 3)  
12  
> (test 1 2 3 4 5)  
30  
> (test 1 2)  
6  
> (test)  
0
```

alle argumenten worden
verzameld in een lijst die
gebonden wordt aan de
enige parameter x

Define & Lambda list – verplichte en optionele argumenten

```
(define test  
  (lambda (x y . z)  
    (+ (* 2 x) (* 2 y)  
      (do ((rest z (cdr rest))  
          (result 0 (+ result (* 2 (car rest)))))  
        ((null? rest) result)))))
```

OF

```
(define (test x y . z)  
  idem
```

> (test 1 2)

2

> (test 1 2 3 4 5)

120

> (test 1)

⊗⊗ *test: arity mismatch;*

the expected number of arguments does not match the given number

expected: at least 2

given: 1 arguments....

verwacht minstens 2 argumenten die worden gebonden aan de parameters x en y, de rest van de argumenten wordt verzameld in een lijst die gebonden wordt aan de parameter achter het punt

Een operator toepassen op zijn argumenten

kijk naar de types
van de argumenten
en haal de juiste
functie uit de tabel

```
(define (apply-generic op . args)
  (let ((types (map type args)))
    (let ((proc (get op types)))
      (if proc
          (apply proc (map content args))
          (error "operator not defined on these types
-- APPLY-GENERIC" op types))))))
```

pas de functie toe
op de 'naakte'
argumenten

De constructors en selectors via de tabel

```
(define (make-from-rect x y)
  ((get 'make-from-rect 'rectangular) x y))
(define (make-from-polar r a)
  ((get 'make-from-polar 'polar) r a))

(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))
```

Les 8: Generische operatoren

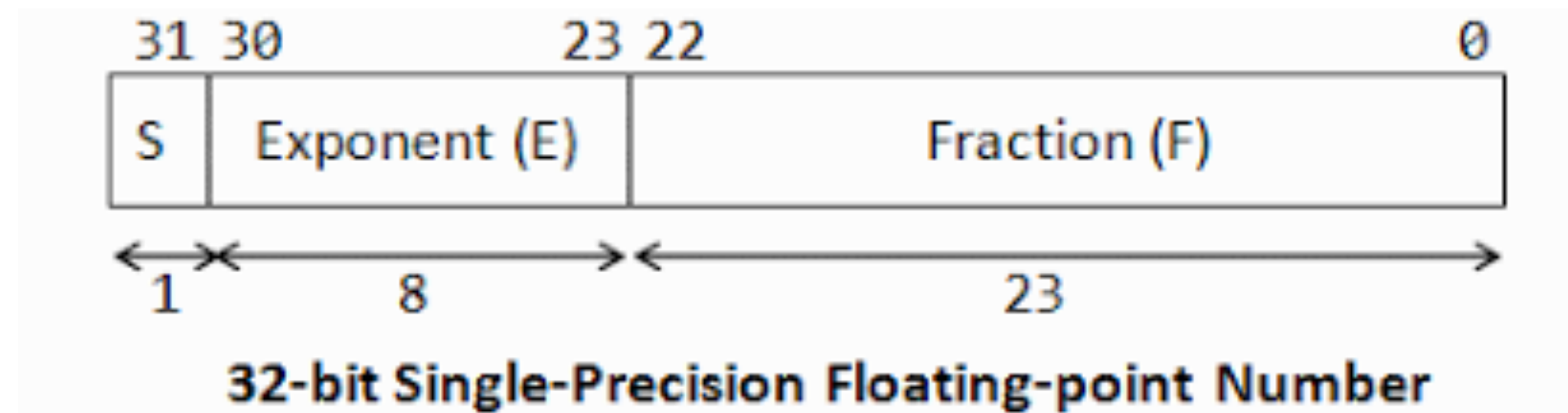
Deel 2

De rekenkundige operatoren in Scheme zijn generisch operatoren

```
10001010010101010110
01001100010001010001
01010000101001010011
10011010010000001010
01010010010010010100
10010010101010101100
10101010000100010011
```

For 25;
Sign bit → 0000000000000000000000000000000011001
31 bits

For (-25)
Sign bit → 1111111111111111111111111111111100110
31 bits



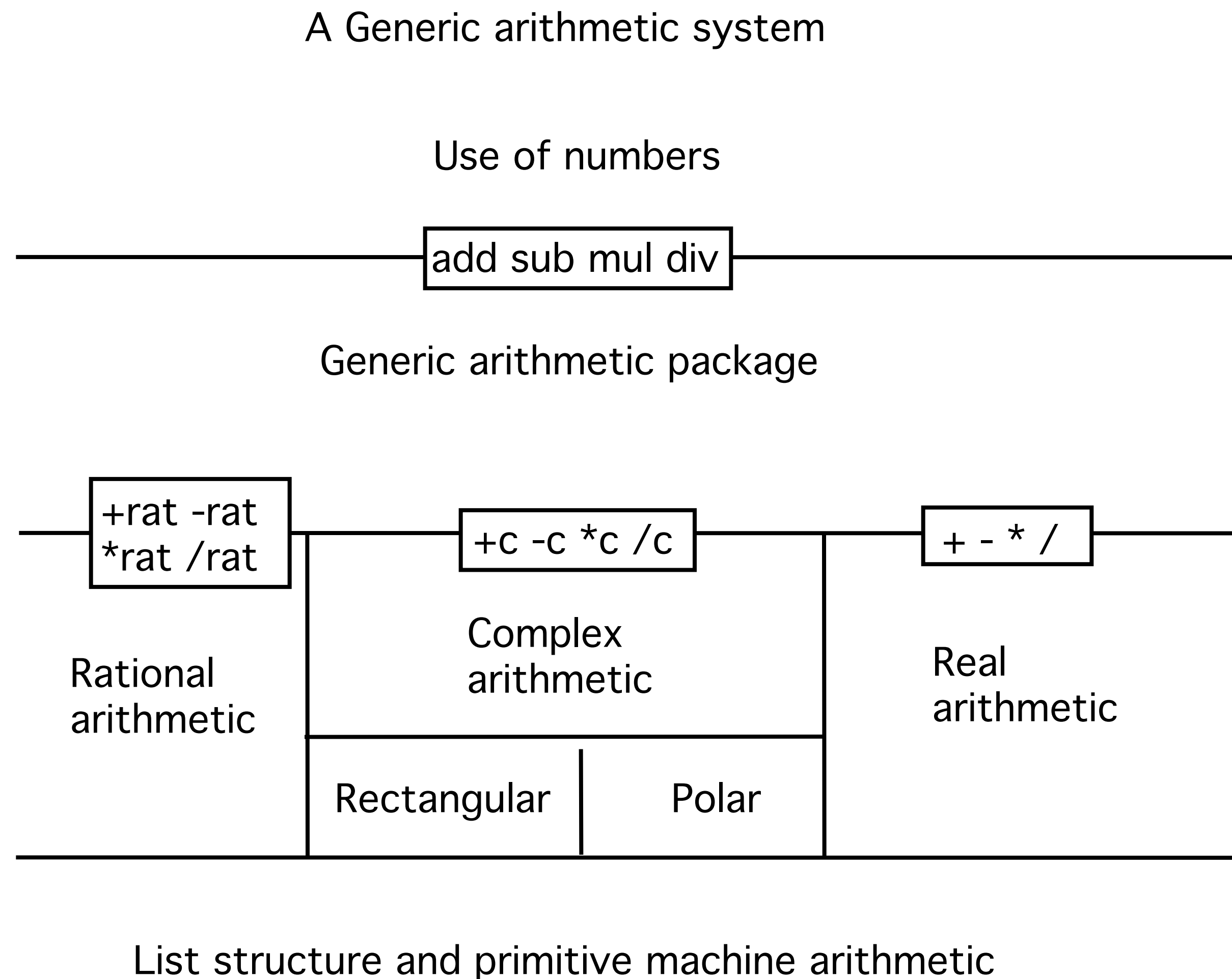
Systeem met generische operatoren

In wiskunde:

- integer
- rational
- real
- complex


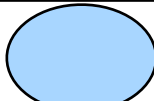














In Scheme:

- scheme-number
 - integer
 - float
- rational
- complex
 - rectangular
 - polar



Generische operatoren: uitwerking

Assume existence of : (put <op> <type> <item>)
(get <op> <type>)

	(scheme scheme)	(rational rational)	(complex complex)	scheme	rational	complex
add						
sub						
mul						
div						
make						
make-from-polar						
make-from-rect						

Interfacing van het ingebouwde Number pakket

```
(define (install-scheme-number-package)
  (define (tag x)
    (attach-type 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
    (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
    (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
    (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
    (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number
    (lambda (x) (tag x)))
  'done)
```

Interfacing van Breuken pakket

```
(define (install-rational-number-package)
  ;;; internal procedures
  (define (make-rat a b) (cons a b))
  (define (denom z) (cdr z))
  (define (numer z) (car z))
  (define (rat+ p q)
    (make-rat (+ (* (numer p) (denom q))
                  (* (numer q) (denom p)))
              (* (denom p) (denom q))))
  (define (rat- p q)
    (make-rat (- (* (numer p) (denom q))
                  (* (numer q) (denom p)))
              (* (denom p) (denom q))))
  (define (rat* p q)
    (make-rat (* (numer p) (numer q))
              (* (denom p) (denom q))))
  (define (rat/ p q)
    (make-rat (* (numer p) (denom q))
              (* (denom p) (numer q))))
```

```
;;; interfacing
(define (tag x)
  (attach-type 'rational x))
(put 'add '(rational rational)
     (lambda (x y) (tag (rat+ x y))))
(put 'sub '(rational rational)
     (lambda (x y) (tag (rat- x y))))
(put 'mul '(rational rational)
     (lambda (x y) (tag (rat* x y))))
(put 'div '(rational rational)
     (lambda (x y) (tag (rat/ x y))))
(put 'make 'rational
     (lambda (a b)
       (tag (make-rat a b))))
'done)
```

Interfacing van Complex pakket

```
(define (install-complex-number-package)
  ;; imported from rect and polar
  (define (make-from-rect x y)
    ((get 'make-from-rect 'rectangular) x y))
  (define (make-from-polar x y)
    ((get 'make-from-polar 'polar) x y))
  ;; internal procedures
  (define (+c z1 z2)
    (make-from-rect
      (+ (real-part z1) (real-part z2))
      (+ (imag-part z1) (imag-part z2))))
  (define (*c z1 z2)
    (make-from-polar
      (* (magnitude z1) (magnitude z2))
      (+ (angle z1) (angle z2))))
  (define (-c z1 z2)
    (make-from-rect
      (- (real-part z1) (real-part z2))
      (- (imag-part z1) (imag-part z2))))
  (define (/c z1 z2)
    (make-from-polar
      (/ (magnitude z1) (magnitude z2))
      (- (angle z1) (angle z2))))
```

```
;; interfacing
(define (tag x) (attach-type 'complex x))
(put 'add '(complex complex)
  (lambda (x y) (tag (+c x y))))
(put 'sub '(complex complex)
  (lambda (x y) (tag (-c x y))))
(put 'mul '(complex complex)
  (lambda (x y) (tag (*c x y))))
(put 'div '(complex complex)
  (lambda (x y) (tag (/c x y))))
(put 'make-from-rect 'complex
  (lambda (x y)
    (tag (make-from-rect x y))))
(put 'make-from-polar 'complex
  (lambda (x y)
    (tag (make-from-polar x y))))
'done)
```

Systeem met generische operatoren

```
(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))

(define (make-scheme-number n)
  ((get 'make 'scheme-number) n))
(define (make-rational a b)
  ((get 'make 'rational) a b))
(define (make-complex-from-rect x y)
  ((get 'make-from-rect 'complex) x y))
(define (make-complex-from-polar r a)
  ((get 'make-from-polar 'complex) r a))
```

Coercion (1)

De implementatie tot nu toe laat alleen toe dat 2 getallen van hetzelfde type worden gebruikt in een bewerking. Om ook toe te laten van een breuk bij een complex getal op te tellen of een complex getal te delen door een gewoon getal etc. zijn er twee opties:

(1) alle operatoren implementeren voor alle mogelijk combinaties van types van getallen en de tabel met operatoren dus veel groter maken

(2) enkele coercion operatoren implementeren die een getal van een gegeven type kunnen omvormen (promoveren) tot een getal van een 'hoger' type en in het geval van twee verschillende types van input, één van de twee promoveren zodat de types van de inputs gelijk worden

Uit wiskunde:

- een geheel getal is ook een breuk noemer 1
- een breuk is ook een reëel getal reken gewoon de deling uit
- een reëel getal is ook een complex getal imaginair deel 0

Coercion (2)

```
(define (rational->number r))  
  (define (denom z) (cdr z))  
  (define (numer z) (car z))  
  (make-scheme-number  
    (/ (numer (content r)) (denom (content r)))))  
  
(define (number->complex n))  
  (make-complex-from-rectangular (content n) 0))  
  
(put-coercion 'rational 'scheme-number  
  rational->number)  
  
(put-coercion 'scheme-number 'complex  
  number->complex)
```

plaats die coercion operatoren
in een aparte kleine tabel

Coercion (3)

```
(define (apply-generic op . args)
  (let ((types (map type args)))
    (let ((proc (get op types)))
      (if proc
          (apply proc (map content args))
          (if (= (length args) 2)
              (let ((co1 (get-coercion (car types) (cadr types))))
                (if co1
                    (apply-generic op (co1 (car args)) (cadr args))
                    (let ((co2 (get-coercion (cadr types) (car types))))
                      (if co2
                          (apply-generic op (car args) (co2 (cadr args)))
                          (error "operator not defined on these types
                                -- APPLY-GENERIC" op types))))))
              (error "operator not defined on these types
                    -- APPLY-GENERIC" op types))))))
```

coercion uitgewerkt voor
2 argumenten

Data <> Procedure – Intro

Zoals reeds vroeger werd uitgelegd bestaan er in de meeste programmeertalen twee soorten objecten: procedure-objecten en data-objecten. In Scheme is het ook zo dat beide soorten objecten evenwaardig zijn, i.e. ook procedure-objecten zijn eerste klas objecten. Maar het is niet zo dat er een soort natuurlijke strikte grens bestaat tussen data-objecten en procedure-objecten.

Er wordt eerst getoond hoe een paar, een op het eerste zich toch wel 'echt' data-object ook als een procedure-object kan geïmplementeerd worden.

Daarna doen we hetzelfde voor de complexe getallen en laten dan zien hoe het implementeren van generische operatoren op deze manier zeer eenvoudig wordt.

Data <> Procedure

```
> (define test (cons 3 4))
> (car test)
3
> (cdr test)
4
> test
(3 . 4)
```

een gewone cons cel

```
(define (my-cons x y)
  (define (dispatch m)
    (cond
      ((eq? m 'car) x)
      ((eq? m 'cdr) y)
      (else (error "unknown operator"))))
  dispatch)

(define (my-car z) (z 'car))

(define (my-cdr z) (z 'cdr))
```

gedraagt zich als een
cons cel

```
> (define test (my-cons 3 4))
> (my-car test)
3
> (my-cdr test)
4
> test
#<procedure:dispatch>
```

Generische Selectoren: Message Passing

```
(define (my-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 operator —
                        MAKE-RECT" m))))
  dispatch)

(define (my-make-polar r a)
  (define (dispatch m)
    (cond ((eq? m 'real-part) (* r (cos a)))
          ((eq? m 'imag-part) (* r (sin a)))
          ((eq? m 'magnitude) r)
          ((eq? m 'angle) a)
          (else (error "unknown operator —
                        MAKE-POLAR" m))))
  dispatch)
```

```
> (define z1
    (my-make-rectangular 1 2))
> (define z2
    (my-make-polar 3 4))
> z1
#<procedure:dispatch>
> z2
#<procedure:dispatch>
> (eq? z1 z2)
#f
> (z1 'real-part)
1
> (z1 'angle)
1.1071487177940904
> (z2 'real-part)
-1.960930862590836
> (z2 'magnitude)
3
```

Message Passing: syntactische suiker

```
(define (send obj op) (obj op))
```

```
> (send z1 'real-part)  
1  
> (send z1 'angle)  
1.1071487177940904
```

```
(define (ask op obj) (obj op))
```

```
> (ask 'real-part z1)  
1  
> (ask 'angle z1)  
1.1071487177940904
```

Les 9: 'Assignment' en lokale toestand
