§3 Programowanie z kombinatorami

Przykład:

- `sumi a b = if a > b then 0`
  `else a + sumi (a+1) b`

- $\frac{1}{8}\pi = \frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \ldots$

```
sumpi = if a > b then 0
        else 1/(a * (a+2)) +
             sumpi (a+4) b
```

- ↳

```
sum term next a b =
    if a > b then 0
    else (term a) +
         sum term next (next a) b
```

oraz

```
sumi = sum (\x -> x) (\x -> x+1)

sumpi = sum (\x -> 1/(x * (x+2)))
            (\x -> x+4)
```

3.1 Funkcje fold

```
sum []     = 0
sum (x:xs) = x + sum xs
```

```
sum = foldr (+) 0
```

```
prod []     = 1
prod (x:xs) = x * prod xs
```

```
prod = foldr (*) 1
```

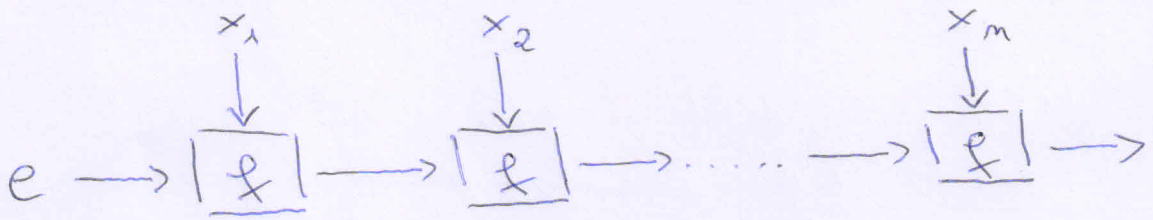
```
reverse []     = []
reverse (x:xs) = (rev xs) ++ [x]
```

```
reverse = foldr ? []
```

```
foldl f e []     = e
foldl f e (x:xs) =
        foldl f (f e x) xs
```

czyli dla $[x_1, \ldots x_n]$

$$f(\ldots f(f(e, x_1), x_2)\ldots, x_n)$$




```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Przykład:

```
sum = foldl (+) 0
```

```
sum :: [a] → a
```

```
sum [1,2,3]
→ foldl (+) 0 [1,2,3]
→ foldl (+) (0+1) [2,3]
→ foldl (+) ((0+1)+2) [3]
→ foldl (+) (((0+1)+2)+3) []
→ ((0+1)+2)+3
```

```
reverse = foldl ? []
```

1. próba: ? = (:)

```
reverse [1,2,3]
→ foldl (:) [] [1,2,3]
→ foldl (:) ([]:1) [2,3]
```

błąd

↳ inne folde?

nie potrzebne:

potrzebujemy 1:[]

czyli f x e zamiast f e x

↳ flip

```
flip f x y = f y x
```

tesa2:

```
reverse = foldl (flip (:)) []
```

```
reverse [1,2,3]
-> foldl (flip (:)) [] [1,2,3]
-> foldl (flip (:)) ((flip (:)) [] 1) [2,3]
                    \_____________/
                        1 : []
-> foldl (flip (:)) ((flip (:)) [1] 2) [3]
*-> [3,2,1]
```

3.2 Divide-and-Conquer

Problem a

```
test :: a -> Bool
end :: a -> b
divide :: a -> [a]
combine :: [b] -> b
```

```
dc test end divide combine p =
    if test p
    then end p
    else combine
           (map (dc test end divide comb)
                (divide p))
```

Prozvlečod:

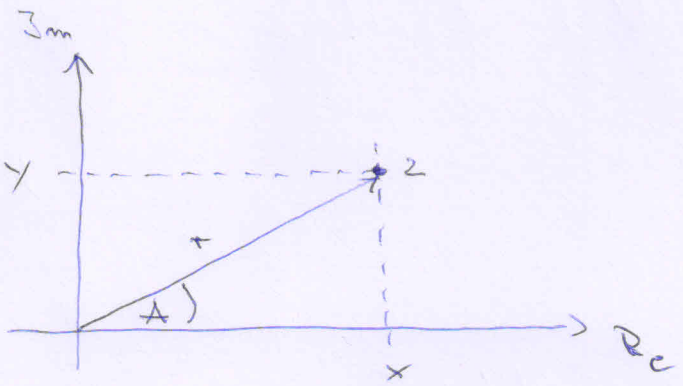
```
quicksort =
  dc (\x -> (length x ≤ 1)
     (\x -> x)
     split
     flatten

split [] = []
split (a:as) =
  [[b | b ← as, b < a], [b | b ← as, b ≥ a] ++ [a]]

flatten [a, b] = a ++ b
```

3.3 Przykład: Liczbe zespolone

$$z = x + iy \quad , \quad i := \sqrt{-1}$$

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

$$z_1 * z_2 = \ldots$$



$$z = r \cdot e^{iA}$$

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

Przeliczenie (zmiana reprezentacji):

$$x = r \cdot \cos A, \ldots$$

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

Realizacja:

$$z = (\text{'r'}, [x, y])$$

albo $z = (\text{'e'}, [r, A])$

'r' i 'e' dają informację, w jakiej reprezentacji jest liczba z.

→ 7

```
-- type Compl = (Char, [Float])


-- plus_c :: Compl -> Compl -> Compl
plus_c z1 z2 =
   make_rectangular ((real_part z1) + (real_part z2))
                    ((imag_part z1) + (imag_part z2))

-- times_c :: (Compl, Compl) -> Compl
times_c (z1, z2) =
   make_polar ((magnitude z1) * (magnitude z2))
              ((angle z1) + (angle z2))


-- make_rectangular :: Float -> Float -> Compl
make_rectangular x y = ('r', [x, y])

-- make_polar :: Float -> Float -> Compl
make_polar x y = ('p', [x, y])


-- real_part :: Compl -> Float
real_part ('r', [a, b]) = a
real_part ('p', [a, b]) = a * (cos b)

-- imag_part :: Compl -> Float
imag_part ('r', [a, b]) = b
imag_part ('p', [a, b]) = a * (sin b)

-- magnitude :: Compl -> Float
magnitude ('r', [a, b]) = a * a + b * b
magnitude ('p', [a, b]) = a

-- angle :: Compl -> Float
angle ('r', [a, b]) = (atan b) * a
angle ('p', [a, b]) = b


-- times_c ((make_rectangular 1 2), (make_polar 3 4))
```

Inna realizacja:

(nie zmieniając funkcje plus_c i times_c)

zmiana reprezentacja w funkcjach make_rectangular i make_polar

↳ liczba z jako funkcja (z argumentem m)
argument m powie jaka informacja (real_part, ...) o liczbie z jest potrzebna.

! ⟶ (9)

Przykład:

real_part (make_polar 1 2)

→ (make_polar 1 2) "real_part"

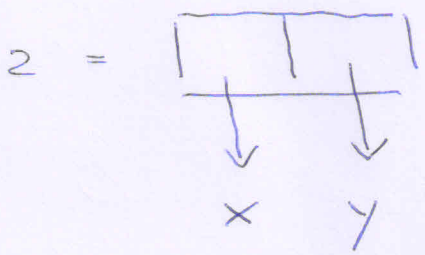
→ (\m → (case m of
"real_part" → 1 * cos 2
"imag_part" → 1 * sin 2
"angle" → 1
"magnitude" → 2)) "real_part"

→ 1 * cos 2

```
-- type Compl = (Char, [Float])

-- plus_c :: Compl -> Compl -> Compl
plus_c z1 z2 =
   make_rectangular ((real_part z1) + (real_part z2))
                    ((imag_part z1) + (imag_part z2))

-- times_c :: (Compl, Compl) -> Compl
times_c (z1, z2) =
   make_polar ((magnitude z1) * (magnitude z2))
              ((angle z1) + (angle z2))


make_polar a b =
  \m -> (case m of
            "real_part" -> a * cos b
            "imag_part" -> a * sin b
            "angle"     -> a
            "magnitude" -> b)

make_rectangular a b =
  \m -> (case m of
            "real_part" -> a
            "imag_part" -> b
            "angle"     -> (atan b) * a
            "magnitude" -> a * a + b * b)

real_part z = z "real_part"
imag_part z = z "imag_part"
angle z = z "angle"
magnitude z = z "magnitude"
```

3.4 Czym są dane?

Para:



→ operator (:)

Własciowości:

1. first (x : y) = x
2. second (x : y) = y
3. (first (x : y)) : (second (x : y)) = x : y

Można rozumieć 1., 2., 3. jako definicję, czym jest para: Jeśli mamy operacje (:), first i second, który spełniają 1., 2., 3. nie musimy wiedzieć jak zostały zrealizowane.

```
cons x y = \m -> m x y
car z   = z (\p q -> p)
cdr z   = z (\p q -> q)
```

cons	" = "	(:)
car	" = "	first
cdr	" = "	second

car (cons x y)

→ (cons x y)(λ p q → p)

→ (λ m → m x y)(λ p q → p)

→ (λ p q → p) x y

→ (λ q → x) y

→ x

↳ cons, car i cdr spełniają 1., 2., 3.

czyli

Zrealowaliśmy pary używając wyłącznie używając funkcje!

↳ nie ma różnicy między danimi i funkcjami.