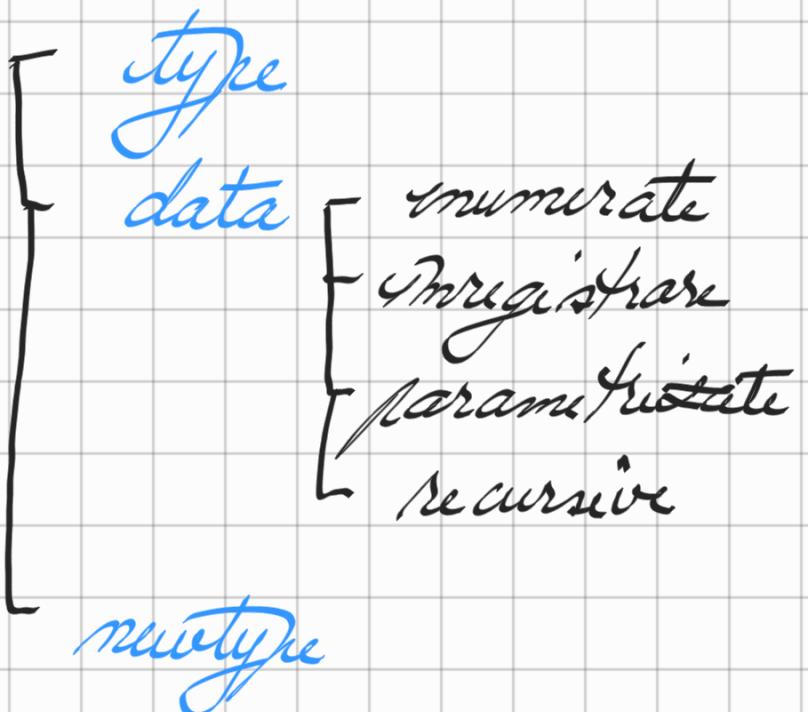


LAB 8: Tipuri de date utilizator

! definiție tipuri



TIPURI SINONIME

("type")

→ se pun un nume unui tip de către utilizator

(=) "typedef" din C

nou tip

Ex:

```
type PhoneRecords = [(String, String)]
```



better

tip de către utilizator

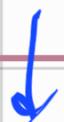
```
type Name = String
type PhoneNumber = String
type PhoneRecords = [(Name, PhoneNumber)]
```

=> type se folosește pentru expresivitate
[(Name, PhoneNumber)] e mai expresivă
decât [(String, String)].

TIPURI DE DATE

- Haskell permite definirea unor noi tipuri
- keyword : "data"

data NumereTip = Constructor1 | Constructor2 | ...



numele tipului
="type constructor"

constructori de date
="value constructors"

folosit în expresii de
tip

folositi pentru
definiții

1 TIPURI ENUMERATE

Ex 1 :

```
data Color = Black | White | Red | Blue | Yellow deriving Show
```

număr
obiect
value constructors

! value constructors sunt fctii care returnează un obiect de tipul asociat

```
*Main> :t Black
Black :: Color
*Main> :t White
White :: Color
```

adică: fctii "fără" parametri

Ex 2: Constructorii cu parametrii

nume tip

1

value constructor cu doi parametrii

```
data Point = Point Float Float deriving Show
```

```
data Shape = Circle Point Float | Rectangle Point Point deriving Show
```

```
point1 = Point 1 2  
point2 = Point 2 3  
circle = Circle point1 3  
rectang = Rectangle point1 point2
```

definire
variabili

value constructor 1

value constructor 2

- ! Constructorii sunt de fapt functii
- ! numele spului si numele constructorului pot coincide, dar nu sunt acelasi lucru

- Point

```
*Main>  
*Main> :t Point  
Point :: Float -> Float -> Point  
*Main> |
```

→ Constructorul **Point** primeste ca parametru două coordonate (Float)

si intorce un obiect de tipul "Point"

- Shape → are doi constructori:

①
 *Main> :t Circle
 Circle :: Point -> Float -> Shape
 *Main>

Constructorul

"Circle"

via ca parametrii punctul
de centru (Point) și raza (Float)

și înțelege un obiect de tip "Shape"

*Main>
 *Main> :t Rectangle
 Rectangle :: Point -> Point -> Shape
 *Main>

Constructorul

"Rectangle"

via ca parametrii
punctele din colț stânga jos + dreapta sus
(Point) și înțelege un obiect "Shape"

! PATTERN HATCHING pe constructori

Ex:

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = abs(x2 - x1) * abs(y2 - y1)
```

Oly:

- Fctii sunt comportament diferit
în funcție de constructor → **PATTERN MATCHING**
 - În funcție este $\boxed{\text{Shape} \rightarrow \text{Float}}$
 - parametru intrare
 - rezultat
 - nu se poate defini fctia ce
În funcție $\text{Circle} \rightarrow \text{Float}$
- "Circle" = constructor, nu fi

Apelarea funcției:

```
*Main> surface $ Circle (Point 10 20) 10
```

```
314.15927
```

```
*Main> surface $ Rectangle (Point 0 0) (Point 100 100)  
10000.0
```

2 TIPURI INREGISTERATE

→ am văzut că suntem într-o situație similară
adică parametrii al unui constructor:

```
data Person = Person String String Int String deriving (Show)  
  
somebody = Person "Buddy" "Finklestein" 43 "526-2928"  
  
firstName :: Person -> String  
firstName (Person firstname _ _ _) = firstname  
  
lastName :: Person -> String  
lastName (Person _ lastname _ _) = lastname  
  
age :: Person -> Int  
age (Person _ _ age _) = age  
  
phoneNumber :: Person -> String  
phoneNumber (Person _ _ _ number) = number
```

} putem construi
functii "getter"
pt. a extrage
completele care
defineste o persoana

→ mai sus, putem folosi inregistrarea:

✓ numele
completui

tipul completului

```
data Person2 = Person2 { firstName2 :: String  
                        , lastName2 :: String  
                        , age2 :: Int  
                        , phoneNumber2 :: String  
} deriving (Show)
```

```
anotherGuy = Person2 "Guy" "Smith" 21 "732658930"
```

! *FirstName2, LastName2, PhoneNumber2, age2* = Setări care accesează campurile ("getters")

Ex:

```
someGuy = Person2 {firstName2 = "Buddy",
                    lastName2 = "Finklestein",
                    age2 = 43,
                    phoneNumber2 = "526-2928"}
```

```
*Main> :t firstName2
firstName2 :: Person2 -> String
*Main> firstName2 someGuy
"Buddy"
```

③ TIPURI PARAMETRIZATE → generic

→ *Setările type constructors* care primesc ca parametru tipuri și generează noi tipuri

Ex:

```
data Maybe a = Nothing | Just a
```

type
constructor

variaabilitate de tip

| (type parameter)

inlocuită cu tip concret

! "Maybe" nu e tip, e constructor de tip

! Maybe Int, Maybe Person ...
→ sunt tipuri

④ TIPURI RECURSIVE

→ constructorul permite ca parametrii
dintre tipul intors ⇒ recursitate

Ex:

```
data List a = EmptyList | Cons a (List a) deriving Show
```

tipul

constructor

constructor cu 2 parametrii

```
l1 = Cons 5 (Cons 6 EmptyList)
```

NEWTYPE

→ Asemānātns ar "data", dar:

- Jēvīgs konstruktoris
- Konstruktores ir jēvīgi campiņi

Ex:
=

```
-- newtype Meters = MakeMeters Double deriving Show  
newtype Meters = MakeMeters {getMeters :: Double} deriving Show  
newtype Feet = MakeFeet {getFeet :: Double} deriving Show  
-- Function to convert meters to feet  
metersToFeet :: Meters -> Feet  
metersToFeet (MakeMeters m) = MakeFeet (m * 3.28084)
```

constructor
parametri