Programare Funcțională

Prof. Gheorghe GRIGORAŞ

www.info.uaic.ro/~grigoras/pf

Cursul 1 - Plan

- Informații generale despre curs
  - Cerințe, Logistica, Bibliografia
- Limbaje de programare
  - Functional vs. Imperativ
  - Pur vs. Impur
  - Lazy vs. Eager
  - Typed vs. Untyped
- Istoric al limbajelor funcționale
- Introducere în Haskell
  - GHC, sesiuni, scripturi
- Tipurile de bază în Haskell
- Tipuri polimorfe, tipuri supraîncărcate

Cerințe

- Curs opțional, anul II, sem II
  - Număr de credite: 5
  - Total ore: 5 x 30 = 150
    - Curs 28
    - Laborator 28
    - Activitate individuală:
      - Antrenament programare Haskell: 28
      - Pregătire teme laborator: 28
      - Parcurgere bibliografie suplimentară 14
      - Pregătire examen scris: 24

- Prezența la toate laboratoarele
  - Evaluare săptămâna 8
  - 1 proiect(1 - 2 persoane) – săpt.9-15
  - 50% din nota finală (nota minimă pentru promovare 5)
- Examen scris la sfârșit 50%

Bibliografie

- https://www.haskell.org
- https://www.haskell.org/documentation
- Graham Hutton, Programming in Haskell, Cambridge 2007, (web)
- Richard Bird, Introduction to Functional Programming using Haskell, Prentice Hall Europe, 1998
- https://www.haskell.org/downloads

Limbaje de programare

- Functional vs. Imperativ
- Pur vs. Impur
- Lazy vs. Eager
- Typed vs. Untyped

Functional vs. Imperativ

- Caracteristici ale stilului funcțional:
  - Structuri de date **persistente**: odată încărcate nu se mai schimbă
  - Metoda primară de calcul: **aplicarea funcțiilor**
  - Structura de control de bază: **recursia**
  - Utilizarea “din greu” a **funcțiilor de ordin înalt**: funcții ce au ca argument alte funcții şi/sau ca rezultat funcții

Functional vs. Imperativ

- Caracteristici ale stilului imperativ:
  - Structuri de date **mutabile**
  - Metoda primară de calcul: **atribuirea**
  - Structura de control de bază: **iterația**
  - Utilizarea **funcțiilor de ordinul întâi**
- Imperativ (C de exemplu):

```
suma = 0;
for(i=1; i<=10; ++i)
  suma = suma + i;
```

- Funcțional:

```
sum[1..10]
```

Quicksort in Haskell

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser)
  ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Quicksort in C

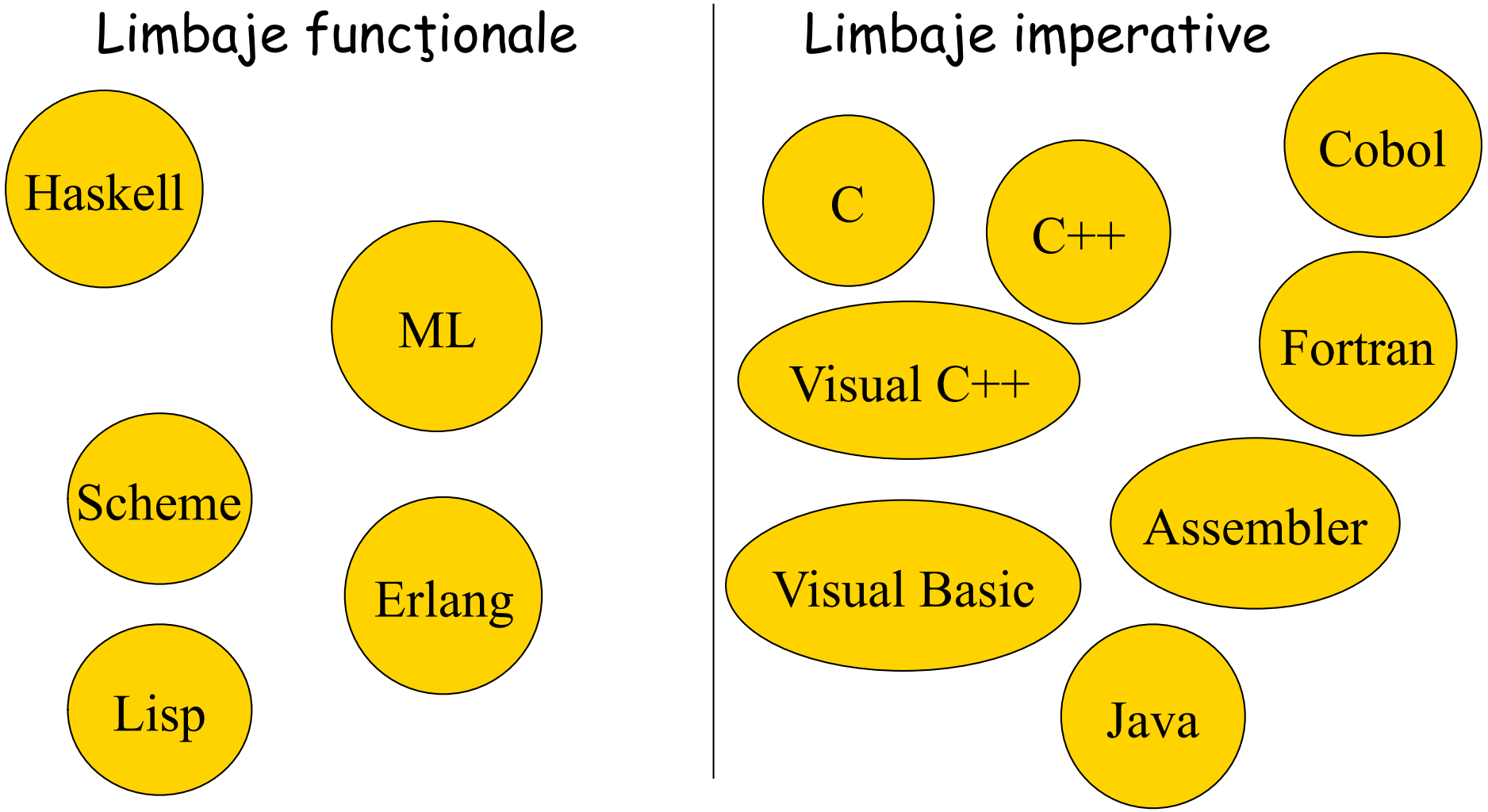
```
// To sort array a[] of size n: qsort(a,0,n-1)
void qsort(int a[], int lo, int hi){
int h, l, p, t;
 if (lo < hi) {
   l = lo;    h = hi;    p = a[hi];
   do {
       while ((l < h) && (a[l] <= p))
              l = l+1;
       while ((h > l) && (a[h] >= p))
              h = h-1;
       if (l < h) {
              t = a[l]; a[l] = a[h]; a[h] = t;
       }
   } while (l < h);
   a[hi] = a[l]; a[l] = p;
   qsort( a, lo, l-1 );
   qsort( a, l+1, hi );
 }
}
```

Avantaje/Dezavantaje

- Functional programs are often easier to **understand.**
- Functional programs tend to be much more **concise, shorter by a factor of two to ten usually, than their imperative counterparts.**
- Polymorphism enhances re-usability: qsort program in Haskell will not only sort lists of integers (*C version)*, but also lists of *anything for which it is meaningful to have "less-than" and "greater-than" operations.*
- Store is allocated and initialized implicitly, and recovered automatically by the garbage collector.

Avantaje/Dezavantaje

- The C quicksort uses an extremely ingenious technique, invented by Hoare, whereby it sorts the array *in place; that is, without using any extra storage.*
- *As a result, it runs quickly, and in a small amount of memory.*
- *In contrast, the Haskell program allocates quite a lot of extra memory behind the scenes, and runs rather slower than the C program.*
- In applications where performance is required at any cost, or when the goal is detailed tuning of a low-level algorithm, an imperative language like C would probably be a better choice than Haskell


Limbaje funcţionale
Haskell
ML
Scheme
Erlang
Lisp
Limbaje imperative
C
C++
Cobol
Visual C++
Fortran
Assembler
Visual Basic
Java

Aplicații industriale implementate funcțional

http://homepages.inf.ed.ac.uk/wadler/realworld/index.html

Intel (microprocessor verification)

Hewlett Packard (telecom event correlation)

Ericsson (telecommunications)

Carlstedt Research & Technology (air-crew scheduling)

Legasys (Y2K tool)

Hafnium (Y2K tool)

Shop.com (e-commerce)

Motorola (test generation)

Thompson (radar tracking)

https://wiki.haskell.org/Haskell

Pur vs. Impur

- Un limbaj funcțional ce are doar caracteristicile stilului funcțional (şi nimic din stilul imperativ) este limbaj funcțional pur. Haskell este limbaj funcțional pur.

- Limbaje ca: Standard ML, Scheme, Lisp combină stilul funcțional cu o serie de caracteristici imperative; acestea sunt limbaje impure.

Lazy vs. Eager

- Limbajele funcționale se împart în două categorii:
  - Cele care evaluează funcțiile (expresiile) în mod **lazy**, adică argumentele unei funcții sunt evaluate numai dacă este necesar
  - Cele care evaluează funcțiile(expresiile) în mod **eager**, adică argumentele unei funcții sunt tratate ca şi calcule precedente funcției şi sunt evaluate înainte ca funcția să poată fi evaluată

- Haskell este un limbaj lazy:

```
Prelude> let f x = 10
Prelude> f(1/0)
10
Prelude> 1/0
Infinity

Prelude> let poz_ints = [1..]
Prelude> let one_to_ten = take 10 poz_ints
Prelude> one_to_ten
[1,2,3,4,5,6,7,8,9,10]
```

Typed vs. Untyped

- Limbajele pot fi categorisite astfel:
  - Puternic tipizate (strongly typed): împiedică programele să acceseze date private, să corupă memoria, să blocheze sistemul de calcul etc.
  - Slab tipizate (weakly typed)
  - Static tipizate (staticaly typed): consistența tipurilor este verificată la compilare
  - Dinamic tipizate (dynamicaly typed): verificarea consistenței se face la execuție

Typed vs. Untyped

- Limbajul Haskell este puternic tipizat
- Orice expresie ce apare într-un program Haskell are un tip ce este cunoscut la compilare (staticaly typed).
- Nu există mecanisme care să “distrugă sistemul de tipuri”.
- Ori de câte ori se evaluează o expresie, de tip întreg de exemplu, rezultatul va fi de acelaşi tip, întreg.

Declarative, statically typed code.

```
primes = filterPrime [2..]
   where filterPrime (p:xs) =
            p : filterPrime [x | x <- xs, x `mod` p /= 0]

primes = filterPrime [2..400]
   where filterPrime (p:xs) =
            p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Haskell este liber de "side effects"!

- Nu are asignări
- Nu are variabile mutabile
- Nu are tablouri mutabile
- Nu are înregistrări mutabile
- Nu are stări ce pot fi actualizate

Avantajele alegerii limbajului Haskell

- Haskell este un limbaj de nivel foarte înalt (multe detalii sunt gestionate automat).
- Haskell este expresiv şi concis (se pot obține o sumedenie de lucruri cu un efort mic).
- Haskell este adecvat în a manipula date complexe şi a combina componente.
- Cu Haskell programatorul câştigă timp: “programmer-time” este prioritar lui “computer-time”

Istoric al limbajelor funcționale

- 1920 – 1940, Alonzo Church(1903 – 1995) şi Haskell Curry(1900 – 1982) dezvoltă “Lambda Calculul”, o teorie matematică a funcțiilor

http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Church.html

Istoric al limbajelor funcționale

- 1970-80, David Turner
  - Limbaje funcționale lazy
  - Limbajul Miranda
- 1987: începe dezvoltarea limbajului Haskell de către un comitet internațional (numele vine de la Haskell Curry) – limbaj funcțional lazy standard
- 2003: este publicat raportul cu definiția limbajului Haskell 98 (“the culmination of fifteen years of revision and extensions”)
- The Haskell 2010 report was published in July 2010, and is the current definition of the Haskell language.

Implementări Haskell

- https://www.haskell.org/downloads#platform
- https://www.haskell.org/platform/
- **Platforma Haskell:**
  - Current release: 2014.2.0.0
  - The Haskell Platform is the easiest way to get started with programming Haskell.
  - The Haskell Platform contains only stable and widely-used tools and libraries, drawn from a pool of thousands of Haskell packages, ensuring you get the best from what is on offer.
  - The Haskell Platform ships with advanced features such as multicore parallelism, thread sparks and transactional memory

Introducere în Haskell; GHC

- GHC, http://hackage.haskell.org/platform/

```
GHCi, version 7.0.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>

GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Sesiuni

- La lansarea sistemului se încarcă o bibliotecă Prelude.hs în care sunt definite funcțiile uzuale:
- Sesiune: secvența de interacțiuni utilizator – sistem

```
Prelude> 56-3*(24+4/2)/(4-2)
17.0
Prelude> 3^21
10460353203
Prelude> 22^33
19950255735593597590945029872666741430235955 2
Prelude> let x = 24 in (x + 14)*(x - 14)
380
Prelude> :{
Prelude| let y = 33
Prelude| in 12 + y*3
Prelude| :}
111
```

Sesiuni

```
Prelude> [1,2,3] ++ [5,6]
[1,2,3,5,6]
Prelude> head [1,2,3,4]
1
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> take 2 [1,2,3,4]
[1,2]
Prelude> drop 2 [1,2,3,4]
[3,4]
Prelude> [1,2,3,4,5] !! 2
3
Prelude> [1,2,3,4,5] !! 9
*** Exception: Prelude.(!!): index too large

Prelude> sum [1..10]
55
```

Scripturi (fișiere sursă)

- O listă de definiții constituie un script:
  - Se editează un fişier cu definiții

```
main = print (fac 20)

fac 0 = 1
fac n = n * fac (n-1)
```

- Se salvează (de exemplu Main.hs)
- Interpretare:
  - Se încarcă acest fişier:

    :load Main.hs sau :l Main.h

  - Se invocă oricare din numele definite în fișier

Scripturi

```
main = print (fac 20)
fac 0 = 1
fac n = n * fac (n-1)
```

```
Prelude> :l Main.hs
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
2432902008176640000
*Main> fac(5)
120
*Main> fac(20)
2432902008176640000
```

Scripturi

```
main = print (fac 20)
fac 0 = 1
fac n = n * fac (n-1)
```

- Compilare:

```
Prelude> :!ghc -o main Main.hs
Prelude> :! main
2432902008176640000
```

Scripturi – modificare + reîncărcare

- Se adaugă la fişierul existent (Main.hs) alte definiții:

```
...
smaller x y = if x < y then x else y
biger x y = if x > y then x else y
```

  - Se salvează
  - Se reîncarcă acest fişier:

```
*Main> :r
Ok, modules loaded: Main.
*Main> biger 55 66
66
*Main> smaller 5 6
5
```

Reguli sintactice

- În Haskell funcțiile au prioritatea cea mai mare: f 2 + 3 înseamnă f(2) + 3
- Apelul funcțiilor se poate face fără a pune argumentele în paranteză:

f(a)	f a
f(x, y)	f x y
f(g(x))	f(gx)
f(x, g(y))	f x (g y)
f(x)g(y)	f x * g y

Reguli sintactice

- Numele funcțiilor şi a argumentelor trebuie să înceapă cu literă mică sau cu _: myF, gMare, x, uu, _x, _y
- Se recomandă ca numele argumentelor de tip listă să aibă sufixul s: xs, ns, us, ps, a13s
- Într-o secvență de definiții toate definițiile trebuie să înceapă pe aceeaşi coloană:

```
a = 30
b = 21
c = 111
```

```
a = 30
 b = 21
 c = 111
```

```
  a = 30
b = 21
  c = 111
```

Cuvinte cheie (21)

Erlang - 28, OCaml - 48, Java - 50,
C++ - 63, Miranda - 10

case	**class**	**data**
default	**deriving**	**do**
else	**if**	**import**
in	**infix**	**infixl**
infixr	**instance**	**let**
module	**newtype**	**of**
then	**type**	**where**

Expresii, Valori

- Expresiile denotă valori
- O expresie poate conține:
  - Numere
  - Valori de adevăr
  - Caractere
  - Tuple(n-uple)
  - Funcții
  - Liste
- O valoare are mai multe reprezentări
- reprezentări ale lui 12:

  12 2+10 3*3+3

Evaluare (reducere, simplificare)

- O expresie este evaluată prin reducerea sa la forme mai simple echivalente
- Evaluatorul limbajului funcțional scrie forma canonică a expresiei care denotă o valoare
- Obținerea formei canonice se face printr-un proces de reducere a expresiilor
- Reducerea se poate face în mai multe moduri: rezultatul trebuie să fie acelaşi

Evaluare (reducere, simplificare)

```
square(2+7)
   def +
square(9)
   def square
9*9
   def *
81
```

```
square(2+7)
   def square
(2+7)*(2+7)
   def +
9*(2+7)
   def +
9*9
   def *
81
```

Evaluare (reducere, simplificare)

- Considerăm scriptul:

```
trei x = 3
infinit = infinit + 1
```

```
trei infinit
    def infinit
trei(infinit + 1)
    def infinit
trei((infinit + 1)+1)
...
```

```
trei infinit
   def trei
3
```

•Strategia lazy evaluation asigură terminarea procesului atunci când acest lucru este posibil

Tipuri

- Tip : colecție de valori împreună cu operațiile ce se pot aplica acestora
  - **`Bool`** conține **`False`** şi **`True`**
  - **`Bool -> Bool`** conține toate funcțiile cu argumente din **`Bool`** şi valori în **`Bool`**
- Se utilizează notația **`v :: T`** pentru a exprima că v are tipul T
- Orice expresie are un tip ce se determină, după reguli precise, înainte de a evalua expresia
- Procesul de determinare a tipului este numit “type inference”

Tipuri

- Tipul unei expresii se poate afla în Haskell:

```
Prelude> :type 4
4 :: (Num t) => t
Prelude> :type 4.54
4.54 :: (Fractional t) => t
Prelude> :type False
False :: Bool
Prelude> :t "alpha" ++ "beta"
"alpha" ++ "beta" :: [Char]
Prelude> :t [1,2,3]
[1,2,3] :: (Num t) => [t]
Prelude> :t ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]
```

Tipurile de bază în Haskell

- Bool
  - –Valori: **`False True`**
  - –Operații: **`&& || not`**
- Char
  - –Valori: caractere
  - –Informații despre tipul char:

```
Prelude> :i Char
```

Tipurile de bază în Haskell

- Operații ale tipului Char

```
Prelude> 'a' == 'b'
False
Prelude> 'a' /= 'b'
True
Prelude> 'a' < 'b'
True
Prelude> 'a' >= 'b'
False
Prelude> succ 'a'
'b'
Prelude> pred 'a'
'`'
Prelude> pred 'b'
'a'
Prelude> toEnum 57 :: Char
'9'
Prelude> fromEnum 'b'
98
```

Tipurile de bază în Haskell

- String
  - Valori: liste de caractere
  - Operații: (vezi liste)

```
Prelude> "99" ++ "a"
"99a"
Prelude> head "alpha"
'a'
Prelude> take 4 "abcdef"
"abcd"
Prelude> drop 4 "abcdef"
"ef"
```

Tipurile de bază în Haskell

- Int: $-2^{31}..2^{31}-1$ (sau $-2^{63}..2^{63}-1$ (întregi precizie fixă)

- Integer (întregi precizie arbitrară)

```
Prelude>2^63::Int
-9223372036854775808
Prelude> 2^63::Integer
9223372036854775808
Prelude> 2^200
1606938044258990275541962092341162602522202993
   782792835301376
```

Operații: + - *...

Tipurile de bază în Haskell

- Float, Double
  - Numere reale în simplă(dubla) precizie: 2.2331, +1.0, 3.141
  - Numărul de cifre semnificative:
    - Virgula fixa: 7 – Float, 16 – Double
    - Virgula mobilă: 1+7, 1+16

```
Prelude> sqrt 4565
67.56478372643548
Prelude> sqrt 4565::Float
67.56478
Prelude> sqrt 456554545454::Float
675688.2
Prelude> sqrt 456554545454
675688.2013576973
Prelude> sqrt 45655454545456667777
2.136713704393868 3e10
Prelude> sqrt 45655454545456667777::Float
2.1367138e10
```

Tipul Listă în Haskell

- Lista: o secvență de elemente de acelaşi tip
- Sintaxa pentru acest tip: [T], unde T este tipul elementelor

```
Prelude> :t [1,2,3]
[1,2,3] :: (Num t) => [t]
Prelude> :t [True, True, False]
[True, True, False] :: [Bool]
Prelude> :t ["True","True","False"]
["True","True","False"] :: [[Char]]
Prelude> :t [[1], [1,2,3],[]]
[[1], [1,2,3],[]] :: (Num t) => [[t]]
Prelude> :t [[[]]]
[[[]]] :: [[[a]]]
```

- [] lista vidă, [1], [“unu”], [[]] liste singleton

Tipul n - uplă în Haskell

- O secvență finită de componente de diferite tipuri; componentele, despărțite prin virgulă, sunt incluse în paranteze rotunde
- Reprezentare: $(T_1, T_2, \ldots, T_n)$
- Numărul componentelor = aritatea tuplei
  - Tupla () – tupla vidă, (T) = T
  - Perechi, triplete
  - Nu există restricții asupra tipului componentelor

```
Prelude> :t ()
() :: ()
Prelude> :t (1)
(1):: (Num t) => t
(2)Prelude> :t ('a', "a", False, 3.4)
('a', "a", False, 3.4) :: (Fractional t) => (Char, [Char],
  Bool, t)
```

Tipul funcție în Haskell

- Funcție în sens matematic, de la tipul T_1 la tipul T_2
- **T_1 -> T_2** notează tipul funcțiilor de la T_1 la T_2
- Convenție Haskell: definiția unei funcții este precedată de tipul său:

```
pi    :: Float
pi = 3.14159
square :: Integer -> Integer
square x = x*x
plus  :: (Int, Int) -> Int
plus(x, y) = x + y
Suma :: Int -> Int -> Int
Suma x y = x + y
```

Funcții curry

- O funcție cu n > 1 argumente este definită ca funcție cu un argument ce are ca valori o funcție cu n-1 argumente

```
plusc      :: Int -> (Int -> Int)
plusc x y = x + y
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
twise  ::  (Int -> Int) -> (Int -> Int)
twise f x = f(fx)
twise f = f.f
```

Funcții curry

- Convenții pentru funcții curry
  - "Operatorul" **->** în tipuri are asociativitatea dreapta

`Int -> Int -> Int -> Int -> Int`

înseamnă

`Int -> (Int -> (Int -> (Int -> Int)))`

`mult x y z înseamnă ((mult x)y)z`

Tipuri polimorfe

- Funcţii ce sunt definite pentru mai multe tipuri: lungimea unei liste se calculează indiferent de tipul elementelor listei
- Variabile tip: a, b, c, ...

```
Prelude> :t length
length :: [a] -> Int
Prelude> length "abcdef"
6
Prelude> length [1,2,3,4,5,6,7,8,9]
9
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
Prelude> :t head
head :: [a] -> a
```

- Un tip ce conţine una sau mai multe variabile tip se numeşte tip polimorf

Tipuri supraîncărcate

- Operatori ce se aplică la mai multe tipuri (`+, *, /, <, >, …`)
- Tipul acestora conține variabile tip supuse unor constrângeri
- Astfel de tipuri se numesc tipuri supraîncărcate

```
Prelude> :t (+)
(+) :: (Num a) => a -> a -> a
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t (<)
(<) :: (Ord a) => a -> a -> Bool
Prelude> :t (/=)
(/=) :: (Eq a) => a -> a -> Bool
```

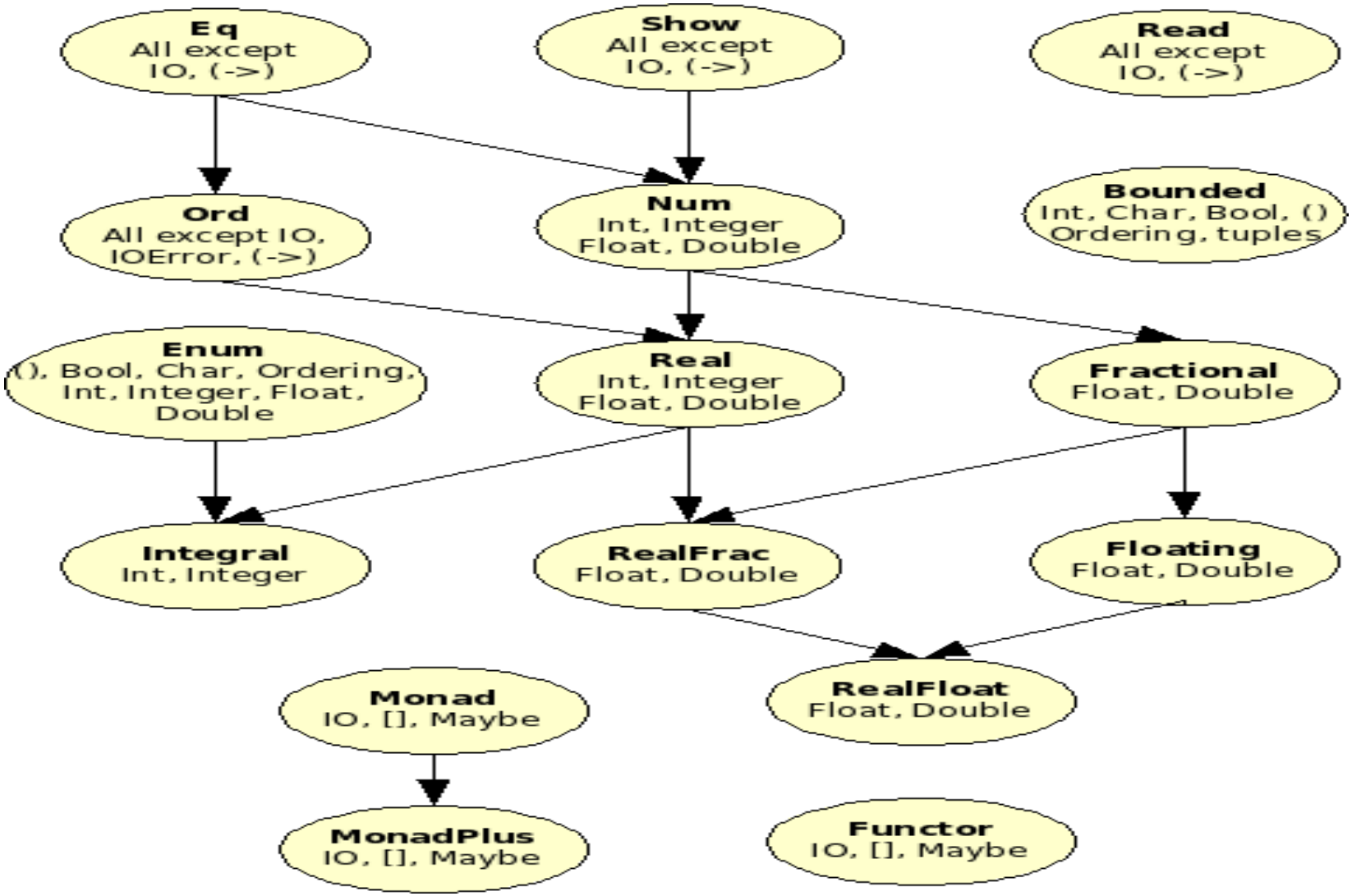
Cursul 2 - Plan

- Clase(de tipuri) în limbajul Haskell
- Definirea funcțiilor
  - Prin folosirea unor funcții existente
  - Expresii condiționale
  - Ecuații cu gardă
  - Potrivire şabloane
    - Şabloane tuple
    - Şabloane listă
    - Şabloane întregi

Clasele de bază

- Tip – colecție de date împreună cu operații
  - Bool, Char, String, Int, Integer, Float
  - Tipul listă
  - Tipul tuplă
  - Tipul funcție
- Clasă – colecție de tipuri care suportă operații supraîncărcate numite *metode*
  - Eq, Ord, Show, Read, Num, Integral, Fractional

Clase în Haskell



Eq – clasa tipurilor cu egalitate

- Tipuri ce au valori care pot fi comparate pentru egalitate şi neegalitate
- Metodele:
  - **`(==) :: Eq a => a -> a -> Bool`**
  - **`(/=) :: Eq a => a -> a -> Bool`**
- Tipuri componente:
  - Tipuri de bază: **`Bool, Char, String, Int, Integer, Float, Double`**
  - Tipuri listă şi tuple cu elemente (componente) tipuri din clasa **`Eq`**

`Ord` – clasa tipurilor ordonate

- Tipuri ce sunt instanțe ale clasei Eq şi care au valorile total ordonate
- Metodele:
  - `(<) :: Ord a => a -> a -> Bool`
  - `(<=) :: Ord a => a -> a -> Bool`
  - `(>) :: Ord a => a -> a -> Bool`
  - `(>=) :: Ord a => a -> a -> Bool`
  - `min :: Ord a => a -> a -> a`
  - `max :: Ord a => a -> a -> a`
- Tipuri componente:
  - Tipuri de bază: **`Bool, Char, String, Int, Integer, Float, Double`**
  - Tipuri listă şi tuple cu elemente (componente) tipuri din clasa **`Ord`**

Exemple

```
Prelude> True > False
True
Prelude> "bababababab" < "abababababa"
False
Prelude> ['a', 'b', 'c'] >= ['x']
False
Prelude> (1,2,3) < (1,2,4)
True
Prelude> min [1,2,-4,5] [1,2,3]
[1,2,-4,5]
Prelude> max (True, False) (False, False)
(True,False)
Prelude> sqrt 777 >= 44
False
```

Show

- Tipuri ce conțin valori exprimabile prin şiruri de caractere
- Metodele:
  - **`show :: Show a => a -> String`**
- Tipuri componente:
  - Tipuri de bază: **`Bool, Char, String, Int, Integer, Float, Double`**
  - Tipuri listă şi tuple cu elemente (componente) tipuri din clasa **`Show`**

Exemple

```
Prelude> show 76890
"76890"
Prelude> show [[12,33],[1],[1,2,3]]
"[[12,33],[1],[1,2,3]]"
Prelude> show ("True", True)
"(\"True\",True)"
Prelude> show (True, True)
"(True,True)"
Prelude> show ('a',777)
"('a',777)"
Prelude> show (44+99)
"143"
```

Read – clasa tipurilor ce pot fi “citite”

- Tipuri ce conțin valori care pot fi convertite din şiruri de caractere
- Metodele:
  - **`read :: Read a => String -> a`**
- Tipuri componente:
  - Tipuri de bază: **`Bool, Char, String, Int, Integer, Float, Double`**
  - Tipuri listă şi tuple cu elemente (componente) tipuri din clasa **`Read`**

Exemple

```
Prelude> read "123"::Int
123
Prelude> read "123"+ read "123"
246
Prelude> read "123.65" ::Float
123.65
Prelude> read "123.65" ::Int
*** Exception: Prelude.read: no parse
Prelude> read "123.65" ::Double
123.65
Prelude> read "[1, 2]" ::[Int]
[1,2]
Prelude> read "[True, False, True, True]"::[Bool]
[True,False,True,True]

Prelude> read "abcd"::String
"*** Exception: Prelude.read: no parse
Prelude> read "\"abcd\""::String
"abcd"
```

Num – clasa tipurilor numerice

- Tipuri ce sunt instanțe ale claselor **Eq** și **Show** şi au valori numerice (**Int, Integer, Float, Double**)
- Metodele:

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a
abs :: Num a => a -> a
signum :: Num a => a -> a
fromInteger :: Integer -> a
```

Exemple

```
Prelude> fromInteger 33 :: Integer
33
Prelude> fromInteger 33 :: Double
33.0
Prelude> let x = fromInteger 22::Double
Prelude> x
22.0
Prelude> :t x
x :: Double
Prelude> fromInteger 33333333333333333 :: Int
553997653
Prelude> fromInteger 33333333333333333 :: Double
3.3333333333333e19
```

Real– clasa tipurilor numerice

- Tipurile **Int, Integer, Float, Double**
- Cum se definește:

```
Prelude> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
        -- Defined in GHC.Real
instance Real Integer -- Defined in
  GHC.Real
instance Real Int -- Defined in GHC.Real
instance Real Double -- Defined in
  GHC.Float
instance Real Float -- Defined in
  GHC.Float
```

Exemple

```
Prelude> toRational 12
12 % 1
Prelude> toRational 12.4
6980579422424269 % 562949953421312
Prelude> toRational 1.5
3 % 2
Prelude> toRational 3
3 % 1
Prelude> toRational 3.5
7 % 2
Prelude> toRational (12.4::Float)
6501171 % 524288
Prelude>  (12.4::Float)
12.4
Prelude> toRational (12.4::Double)
6980579422424269 % 562949953421312
```

`Integral` – clasa tipurilor întregi

- Tipuri **`Int, Integer`**
- Metodele:

```
div :: Integral a => a -> a -> a
mod :: Integral a => a -> a -> a
```

- Metodele pot fi utilizate şi ca operatori infix dacă se scriu **`` `div` ``**, **`` `mod` ``** (atenție: ` şi nu ’)

Exemple

```
Hugs.Base> 22 `mod` 5
2
Hugs.Base> 22 `mod` 3
1
Hugs.Base> 22 `div` 3
7
Hugs.Base> div 22 3
7
Hugs.Base> mod 22 3
1
```

`Fractional` – clasa tipurilor reale

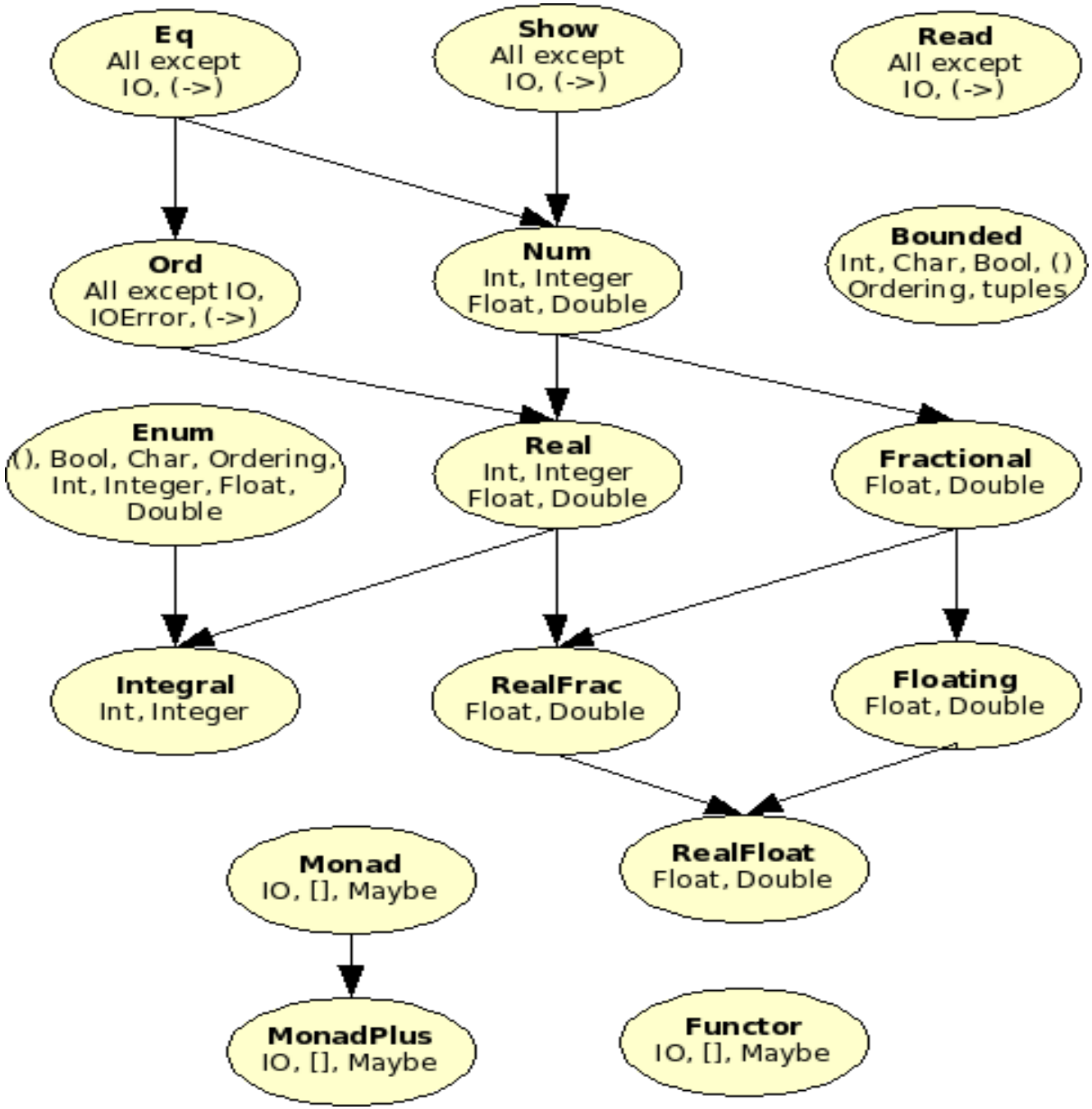
- Tipuri ce sunt instanțe ale clasei **`Num`** şi au valori neîntregi (**`Float, Double`**)

- Metodele:

```
(/) :: Fractional a => a -> a -> a
recip :: Fractional a => a -> a
```

Exemple

```
Hugs.Base> 7/2
3.5
Hugs.Base> 7./2
ERROR - Undefined variable "./"
Hugs.Base> 7.0/2
3.5
Hugs.Base> 7.3/2
3.65
Hugs.Base> recip 2
0.5
Hugs.Base> recip 1
1.0
Hugs.Base> recip 1.5
0.666666666666667
```


Eq
All except
IO, (->)
Show
All except
IO, (->)
Read
All except
IO, (->)
Ord
All except IO,
IOError, (->)
Num
Int, Integer
Float, Double
Bounded
Int, Char, Bool, ()
Ordering, tuples
Enum
(), Bool, Char, Ordering,
Int, Integer, Float,
Double
Real
Int, Integer
Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

Definirea funcțiilor

- Prin folosirea unor funcții existente:

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'

estePar :: Integral a => a -> Bool
estePar n = n `mod` 2 == 0

rupeLa :: Int -> [a] -> ([a], [a])
rupeLa n xs = (take n xs, drop n xs)
```

Exemple

```
Prelude> :l c2test
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> isDigit 'g'
False
*Main> isDigit '8'
True
*Main> even 44
*Main> estePar 56
True
*Main> rupeLa 3 [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

Exemple

- Pot fi definite funcții și ”on-line”:

```
Prelude> let semiP a b c = (a+b+c)/2
Prelude> let ariaT p a b c = sqrt  (p*(p-a)*(p-b)*(p-c))
Prelude> ariaT (semiP 3 4 5) 3 4 5
6.0

Prelude> let x1 a b c = (-b+sqrt(b*b-4*a*c))/(2*a)
Prelude> let x2 a b c = (-b-sqrt(b*b-4*a*c))/(2*a)
Prelude> x1 1 5 6
-2.0
Prelude> x2 1 5 6
-3.0

Prelude> let radacini a b c = let {disc = sqrt(b*b-4*a*c);doia =
  2*a} in ((-b+disc)/doia, (-b-disc)/doia)
Prelude> radacini 1 2 1
(-1.0,-1.0)
Prelude> radacini 1 12 35
(-5.0,-7.0)
```

Definirea funcțiilor

- Expresii condiționale:

  `if` *condiție* **`then`** *res1* **`else`** *res2*

  – *res1* şi *res2* sunt expresii de acelaşi tip
  – Nu există **`if`** fără **`else`** în Haskell

```
semn::Int -> Int
semn n = if n < 0 then -1 else
                 if n == 0 then 0 else 1
```

Definirea funcțiilor

- Expresii case:

```
f1 x =
   case x of
      0 -> 1
      1 -> x + 2
      2 -> x * 3
      _ -> -1
f2 x = case x of
  {0 -> 10; 1 -> 2*x; 2 -> x*x+2; _ -> 0}

Case> f1 1
3
Case> f2 2
6
```

```
Prelude> :{
Prelude| take m ys  = case (m,ys) of
Prelude|                  (0,_) -> []
Prelude|                  (_,[])-> []
Prelude|                  (n,x:xs) ->  x :
                                       take (n-1) xs
Prelude| :}
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
Prelude> take 8 [1,2,3,4,5,6]
[1,2,3,4,5,6]
Prelude> take 0 [1,2,3,4,5,6][]Prelude>
```

Definirea funcțiilor

```
Prelude> let mySign x = if x < 0 then -1 else if x == 0 then 0
   else 1
Prelude> mySign 9
1
Prelude> mySign (-99)
-1
Prelude> mySign 0
0

Prelude> let f x = case x of { 1 -> 1;2 -> x * x;3 -> x+x;4 -> 0;_
   -> 10}
Prelude> f 5
10
Prelude> f 3
6
Prelude> f 2
4
Prelude> f 0
10
```

Definirea funcțiilor

- Ecuații gardate (cu gărzi)
  - Alternativă la condiții cu if
  - Secvență de expresii logice: gărzi
  - Secvență de expresii (rezultate) de acelaşi tip
  - Sintaxa:

nume_func parametri | *garda1* = *exp1*
| *garda2* = *exp2*
…
| *gardan* = *expn*

  - Ultima gardă poate fi **`otherwiswe`** (definită în bibliotecă cu valoarea **`True`**)

Exemple

```
semng n       | n < 0       = -1
              | n == 0      = 0
              | otherwise = 1

*Main> semng 7

<interactive>:1:0: Not in scope: `semng'
*Main> :r
[1 of 1] Compiling Main      ( c2test.hs, interpreted )
Ok, modules loaded: Main.
*Main> semng 7
1
*Main> semng -97

<interactive>:1:0:
    No instance for (Num (a -> a1))
      arising from a use of `-' at <interactive>:1:0-8
    Possible fix: add an instance declaration for (Num
   (a -> a1))
    In the expression: semng - 97
    In the definition of `it': it = semng - 97
*Main> semng (-97)
-1
```

Definiții locale

- O funcție care are în expresia sa o frază de forma “where ...” spunem că are o definiție locală; definiția din fraza where este valabilă doar local
- Exemplu: f(x,y) = (a+1)(a+2), unde a = (x+2)/3

```
f :: (Float, Float) ->Float
f(x,y) = (a+1)*(a+2) where a = (x+2)/3

f :: (Float, Float) ->Float
f(x,y) = (a+1)*(b+2)
         where a = (x+y)/3
               b = (x+y)/2
```

Definiții locale

```
g  :: Integer -> Integer -> Integer
g x y      | x <= 10 = x + a
           | x > 10 = x - a
           where a = square(y+1)
```

- Clauza **`where`** califică ambele ecuații gardate

```
*Main> g 3 4
28
*Main> f (5,6)
35.0
*Main> g 11 2
2
```

```
Prelude> let aria_triunghi a b c =
  sqrt(p*(p-a)*(p-b)*(p-c)) where p =
  (a+b+c)/2

Prelude> aria_triunghi 3 4 5
6.0
Prelude> aria_triunghi 35 35 35
530.4405598179686
```

Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”)
  - Secvență de şabloane – rezultate
  - Sintaxa:

numefunc :: tipul_funcției
Şablon1 = exp1
Şablon2 = exp2
…
Şablonn = expn

  - Dacă se potriveşte primul şablon, se alege ca rezultat exp1, dacă nu se încearcă al doilea şablon, etc.
  - Şablonul _ (“wildcard”) poate fi folosit pentru a exprima potrivirea cu orice valoare

Exemple

```
non :: Bool -> Bool
non False = True
non True = False

conj :: Bool -> Bool -> Bool
conj True True   = True
conj True False  = False
conj False True  = False
conj False False = False

(&) :: Bool -> Bool -> Bool
True & True = True
_ & _ = False
```

Exemple

```
*Main> :type non
non :: Bool -> Bool
*Main> :type conj
conj :: Bool -> Bool -> Bool
*Main> conj (1<2) (2<11)
True
*Main>  (1<2) `conj` (2<11)
True
*Main> (7-2>5) & (2<9)
False
*Main> (7-2>=5) & (2<9)
True
```

Șabloane vs Gărzi

f p_{11} ... p_{1k} = e_1

...

f p_{n1} ... p_{nk} = e_n

este echivalent cu:

f x1 x2 ... *xk = case (x1, ..., xk) of*

(p_{11}, ..., p_{1k}) -> e_1

...

(p_{n1}, ..., p_{nk}) -> e_n

Exemple

- O altă modalitate de utilizare a şabloanelor (utilă în evaluarea lazy): dacă primul argument este True (la conjuncţie) rezultatul este valoarea celui de-al doilea

```
(&&) :: Bool -> Bool -> Bool
True && b = b
False && _ = False

(||) :: Bool -> Bool -> Bool
False || b = b
True || _ = True
```

Exemple

- Nu se poate folosi numele unui argument de două ori:

```
(&&) :: Bool -> Bool -> Bool
b && b = b
False && _ = False
```

- Soluția: folosirea gărzilor:

```
(&) :: Bool -> Bool -> Bool
b & c      | b == c  = b
           | otherwise = False
```

Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”):
  - Şabloane de tip tuple: o tuplă de şabloane este un şablon

```
fst :: (a, b) -> a
fst (x, _) = x
snd :: (a, b) -> b
snd (_, y) = y
```

Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”):
  - Şabloane de tip listă: o listă de şabloane este un şablon

```
test3a   :: [Char] -> Bool
test3a ['a', _, _] = True
test3a _ = False

*Main> test3a [ 'a', 'c', 'r']
True
*Main> test3a [ 'a', 'c', 'r', 'a']
False
```

Utilizarea operatorului “:” la liste (cons)

- Orice listă este construită prin repetarea operatorului cons(“:”) care înseamnă adăugarea unui element la o listă
  - Lista [1,2,3,4] înseamnă
    1:(2:(3:(4:[])))

```
test    :: [Char] -> Bool
test ('a': _) = True
test _ = False
```

- Şabloanele listă trebuiesc puse în paranteză

```
*Main> :type test
test :: [Char] -> Bool
*Main> test ['q', 'w', 'e']
False
*Main> test ['a', 'q', 'w', 'e']
True

null :: [a] -> Bool
nul[] = True
nul(_:_) = False

head  :: [a] -> a
head(x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Definirea funcțiilor

- Tehnica şabloanelor (“pattern matching”):
  - Şabloane “întregi”: expresii de forma n+k unde n este o variabilă întreagă iar k este o constantă întreagă pozitivă

```
pred    :: Int -> Int
pred 0        = 0
pred(n+1)     = n
```

- Şabloanele întregi se potrivesc doar cu expresii pozitive
- Şabloanele întregi trebuiesc puse în paranteză