Programare Funcţională

Prof. Gheorghe GRIGORAŞ

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

Cursul 1 - Plan

- Informaţii generale despre curs
  - Cerinţe
  - Logistica
  - Bibliografia
- Limbaje de programare
  - Functional vs. Imperativ
  - Pur vs. Impur
  - Lazy vs. Eager
  - Typed vs. Untyped
- Istoric al limbajelor funcţionale
- Introducere în Haskell

Cerinţe

- Curs opţional, anul II, sem II
  - Număr de credite: 5
  - Total ore: 5 x 30 = 150
    - Curs 28
    - Laborator 28
    - Activitate individuală:
      - Antrenament programare Haskell: 28
      - Pregatire teme laborator: 28
      - Parcurgere bibliografie suplimentară 14
      - Pregătire examen scris: 24
- Prezenţa la toate laboratoarele
  - Prezentare portofoliu programe
  - 1 proiect (1 - 2 persoane)
  - 50% din nota finală
- Examen scris(ultima săptămână a semestrului) 50%

Logistica

- Curs: G. Grigoraş, 404(etaj 2)
  - Pagina cursului:

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

- Laborator: Georgiana Caltais
  - Pagina laboratorului:

    http://thor.info.uaic.ro/~gcaltais/pf/

Bibliografie

- Graham Hutton, Programming in Haskell, Cambridge 2007, (web)
- Richard Bird, Introduction to Functional Programming using Haskell, Prentice Hall Europe, 1998
- Hal Daume III, Yet Another Haskell Tutorial, http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf
- Dan Popa, Introducere in Haskell 98 prin exemple, EduSoft 2007.
- Mihai Gontineac, Programare functionala - O introducere utilizand limbajul Haskell, Iasi 2007

http://www.haskell.org/haskellwiki/Haskell
http://www.haskell.org/haskellwiki/Ro/Haskell
http://en.wikibooks.org/wiki/Haskell

Limbaje de programare

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

Functional vs. Imperativ

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

Functional vs. Imperativ

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

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

- Funcţional:

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

Limbaje funcţionale

Haskell

ML

Scheme

Erlang

Lisp

Limbaje imperative

C

C++

Cobol

Visual C++

Fortran

Assembler

Visual Basic

Java

Aplicaţii industriale implementate funcţional

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)

Pur vs. Impur

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

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

Lazy vs. Eager

- Limbajele funcţionale se împart în două categorii:
  - Cele care evaluează funcţiile(expresiile) în mod **lazy**, adică argumentele unei funcţii sunt evaluate numai dacă este necesar
  - Cele care evaluează funcţiile(expresiile) în mod **eager**, adică argumentele unei funcţii sunt tratate ca şi calcule precedente funcţiei şi sunt evaluate înainte ca funcţia să poată fi evaluată
- Haskell este un limbaj din prima categorie

Typed vs. Untyped

- Limbajele au, în general, noţiunea de tip
- Sistemul de tipuri diferă
- Tipurile sunt utilizate pentru:
  - Numirea şi organizarea conceptelor
  - Asigurarea că secvenţele de biţi din memorie sunt interpretate consistent
  - Furnizarea către compilator a informaţiilor privind manipularea datelor de către program

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): consistenţa tipurilor este verificată la compilare
  - Dinamic tipizate (dynamicaly typed): verificarea consitenţei se face la execuţie

Typed vs. Untyped

- Limbajul Haskell este puternic tipizat
- Orice expresie ce apare într-un program Haskell are un tip ce este cunoscut la compilare.
- 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.

Avantajele alegerii limbajului Haskell

- Haskell este un limbaj de nivel foarte înalt (multe detalii sunt gestionate automat).
- Haskell este expresiv şi concis (se pot obţine 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 funcţionale

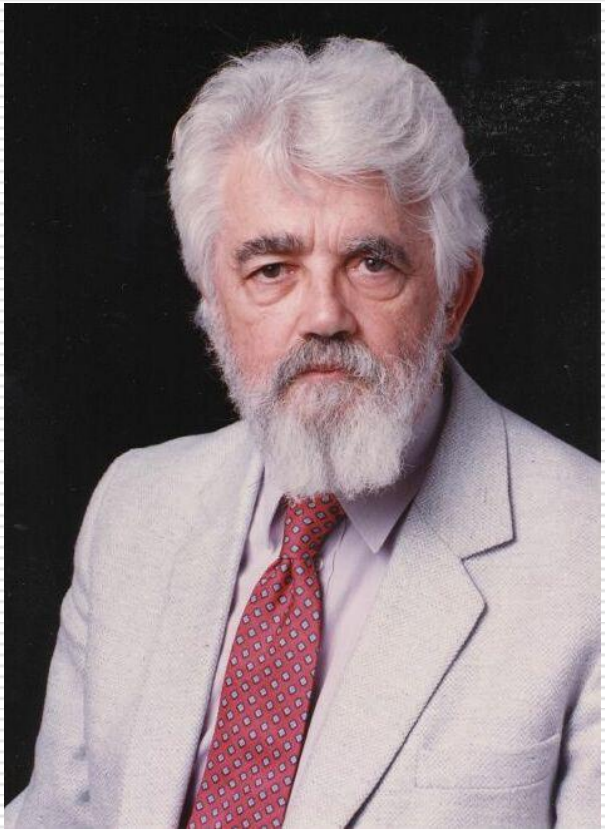
- 1920 – 1940, Alonzo Church(1903 – 1995) şi Huskell Curry(1900 – 1982) dezvoltă "Lamda Calculul", o teorie matematică a funcţiilor

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

Istoric al limbajelor funcţionale

- 1950-1960 - Lisp,
  John McCarthy(n.1927),
- Professor of Computer Science at Stanford University since 1962

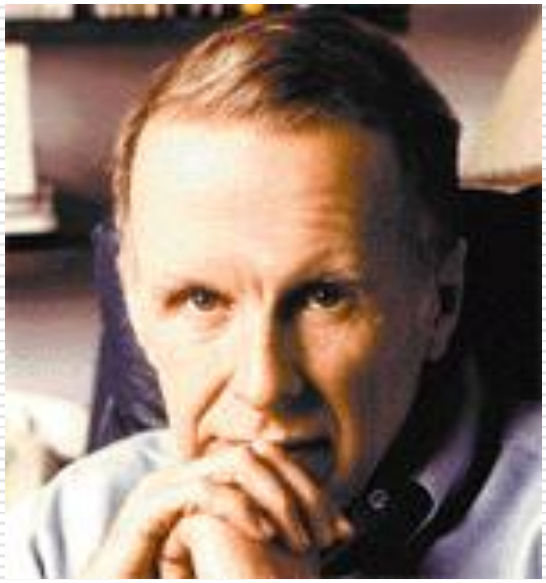
http://www-formal.stanford.edu/jmc/personal.html



Istoric al limbajelor funcţionale

- 1978 - FP,

John Backus (n. 1924)



http://www.thocp.net/biographies/backus_john.htm

- Funcţii de ordin înalt
- Raţionamente asupra programelor

Istoric al limbajelor funcţionale

- ML, Robin Milner (n. 1924)
  - A fost profesor timp de 22 ani la Computer Science Department, University of Edinburgh apoi 4 ani la Cambridge.
  - În 1991 laureat al premiului Turing acordat de ACM (considerat echivalentul premiului Nobel pentru Informatică). Au mai primit acest premiu McCarthy şi Backus
  - Între anii **1971 - 1980**, a condus echipa ce a proiectat limbajul **Standard ML**, unul din primele limbaje care are semantica descrisă formal.
    - Tipurile polimorfe
    - Inferenţa tipurilor

Istoric al limbajelor funcţionale

- 1970-80, David Turner
  - Limbaje funcţionale lazy
  - Limbajul Miranda ("admirable")
- 1987: începe dezvoltarea limbajului Haskell de către un comitet internaţional (numele vine de la Haskell Curry) – limbaj funcţional lazy standard
- 2003: este publicat raportul cu definiţia limbajului Haskell 98 ("the culmination of fifteen years of revision and extensions")

Implementări Haskell

- **Interpreterul Hugs** http://haskell.org/hugs
  - "This small, portable Haskell interpreter written in C runs on almost any machine"
- **WinHugs:** "a rewritten version of WinHugs, and complete Windows libraries"

  http://www-users.cs.york.ac.uk/~ndm/winhugs/
- **Compilatorul GHC,** http://www.haskell.org/ghc/
  - "GHC is a state-of-the-art, open source, compiler... for the Haskell ... works on several platforms including Windows and most varieties of Unix"
- **Visual Haskell** http://www.haskell.org/visualhaskell/
  - "...is a complete development environment for Haskell software, based on Microsoft's Visual Studio platform (Visual Studio .NET 2003 or Visual Studio .NET 2005.)

Introducere în Haskel:

```
Sistemul Hugs
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98
     standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web:
     http://haskell.org/hugs
||   ||                      Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006
     ________________________________________________
     _

Haskell 98 mode: Restart with command line option -98 to enable
     extensions

Type :? for help
Hugs>
```

Introducere în Haskel

```
Hugs> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
.....
:main <aruments>    run the main function with the given arguments
:find <name>        edit module containing definition of name
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Hugs>
```

Sesiuni

- La lansarea sistemului Hugs se încarcă o bibliotecă Prelude.hs în care sunt definite funcţiile cele mai des folosite
- Sesiune: secvenţa de interacţiuni utilizator - sistem

```
Hugs> 78+88*2
254
Hugs> 3^25
847288609443
Hugs> 22^33
19950255735593597590945029872666741430235955 2
```

Sesiuni

```
Hugs> tail [1,2,3,4,5,6]
[2,3,4,5,6]
Hugs> head [1,2,3,4,5,6]
1
Hugs> take 2 [1,2,3,4,5,6]
[1,2]
Hugs> drop 2 [1,2,3,4,5,6]
[3,4,5,6]
Hugs> [1,2,3,4,5,6]!!4
5
Hugs> [1,2,3,4,5,6]!!8

Program error: Prelude.!!: index too large

Hugs> sum[1,2,3,4,5,6]
21
Hugs> sum [1..6]
21
```

Sesiuni

```
Hugs.Base> reverse [1,2,3,4,5,6]
[6,5,4,3,2,1]
Hugs.Base> [1,2,3,4] ++ [5,6,7]
[1,2,3,4,5,6,7]
Hugs> reverse [1..6]
[6,5,4,3,2,1]
Hugs> reverse [1..6]++[3,4]
[6,5,4,3,2,1,3,4]
Hugs> 3:[1,2,3]
[3,1,2,3]
Hugs> fst (22, "hello")
22
Hugs> snd (22, "hello")
"hello"
```

Scripturi

- O listă de definiţii constitue un script:
  - Se editează un fişier cu definiţii

```
module Test
   where
a = 22
b = (3, "hello")
c = a * fst b
double x = x + x
quadruple x = double (double x)
square x = x * x
smaller x y = if x < y then x else y
```

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

```
:load Test.hs sau :l Test.h
```

Scripturi

```
Hugs> :l Test.hs
Test> a
22
Test> b
(3,"hello")
Test> c
66
Test> double 5
10
Test> smaller 3 12
3
Test> smaller (double 3) (square(double 1))
4
Test>
```

Scripturi - modificare + reîncărcare

- Se adaugă la fişierul existent alte definiţii:

```
...
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> :reload
Main> biger 3 7
7
```

Reguli sintactice

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

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

Reguli sintactice

- Numele funcţiilor 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 secvenţă de definiţii toate definiţiile 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

`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 conţine:
  - Numere
  - Valori de adevăr
  - Caractere
  - Tuple(n-uple)
  - Funcţii
  - Liste
- O valoare are mai multe reprezentări: 12 are reprezentări: 12 2+10 3*3+3

Evaluare (reducere, simplificare)

- O expresie este evaluată prin reducerea sa la forme mai simple echivalente
- Evaluatorul limbajului funcţional scrie forma canonică a expresiei care denotă o valoare
- Obţinerea 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 evalation asigură terminarea procesului atunci când acest lucru este posibil

Tipuri

- Tip : colecţie de valori
  - **`Bool`** conţine **`False`** şi **`True`**
  - **`Bool -> Bool`** conţine toate funcţiile cu argumente din **`Bool`** şi valori în **`Bool`**
- Se utilizează notaţia **`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 Haskel:

```
Main> :type 4
4 :: Num a => a
Main> :type False
False :: Bool
Main> :type 5.43
5.43 :: Fractional a => a
Main> :type 5+19
5 + 19 :: Num a => a
Main> :type 5+1.3
5 + 1.3 :: Fractional a => a
```

```
Main> :type "abc"
"abc" :: String
Main> :type not
not :: Bool -> Bool
Main> :type and
and :: [Bool] -> Bool
Main> :type double
double :: Num a => a -> a
Main> :type biger
biger :: Ord a => a -> a -> a
```

Tipurile de bază în Haskel

- Bool
  - Valori: `False True`
  - Operaţii: `&& || not`
- Char
  - Valori: caractere
  - Informaţii despre tipul char:

```
Test> :info Char
-- type constructor
data Char


-- instances:
instance Eq Char
instance Ord Char
instance Enum Char
instance Ix Char
instance Read Char
instance Show Char
instance Bounded Char
```

Tipurile de bază în Haskel

- Operaţii

```
Test> 'a' == 'a'
True
Test> 'a' == 'b'
False
Test> 'a' < 'n'
True
Test> 'b' > 'b'
False
Test> succ 'a'
'b'
Test> pred 'g'
'f'
Test> toEnum 91::Char
'['
Test> toEnum 56::Char
'8'
Test> fromEnum 'b'::Int
98
Test> toEnum 102::Char
'f'
Test>
```

Tipurile de bază în Haskel

- String
  - Valori: liste de caractere
  - Operaţii: vezi liste

```
Test> "99"++"ppp"
"99ppp"
Test> head "oopp"
'o'
Test> drop 3 "facultate"
"ultate"
Test> take 5 " informatica"
" info"
```

Tipurile de bază în Haskel

- Int - $2^{31}..2^{31}-1$ (întregi precizie fixă)
- Integer (întregi precizie arbitrară)

```
Main> 2^31::Int
-2147483648
Main> 2^31
2147483648
Main> 2^31::Integer
2147483648
Main> 2^99
633825300114114700748351602688
```

  - Operaţii: + - *...

Tipurile de bază în Haskel

- Float
  - Numere reale în simplă precizie: 2.2331, +1.0, 3.141
  - Numărul de cifre de la partea zecimală depinde de mărimea numărului

```
Main> sqrt 2
1.4142135623731
Main> sqrt 2::Float
1.414214
Main> sqrt 99999::Float
316.2262
Main> sqrt 9999999
3162.27750205449
Main> sqrt 9999999::Float
3162.278
```

Tipul Listă în Haskel

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

```
Main> :type [1,2,3]
[1,2,3] :: Num a => [a]
Main> :type [True, True, False, False]
[True,True,False,False] :: [Bool]
Main> :type ["True", "True"]
["True","True"] :: [[Char]]
Main> :type ['a','c','d']
['a','c','d'] :: [Char]
Main> :type "unu"
"unu" :: String
Main> :type [['a','c','d'], ['a']]
[['a','c','d'],['a']] :: [[Char]]
```

- [] lista vidă, [1], ["unu"], [[]] liste singleton

Tipul "tuple" în Haskel

- O secvenţă finită de componente de diferite tipuri; componentele, despărţite 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 restricţii asupra tipului componentelor

```
Hugs.Base> :type (1, 'e', 8.3, False)
(1,'e',8.3,False) :: (Fractional a, Num b) => (b,Char,a,Bool)
Hugs.Base> :type ()
() :: ()
Hugs.Base> :type (1)

1 :: Num a => a
```

Tipul funcţie în Haskel

- Funcţie în sens matematic, de la tipul T_1 la tipul T_2
- T_1 `->` T_2 notează tipul funcţiilor de la T_1 la T_2
- Convenţie Haskel: definiţia unei funcţii 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
```

Funcţii curry

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

Funcţii curry

- Convenţii pentru funcţii 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 polimorfice

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

```
Hugs.Base> :type length
length :: [a] -> Int
Hugs.Base> :type fst
fst :: (a,b) -> a
Hugs.Base> :type zip
zip :: [a] -> [b] -> [(a,b)]
Hugs.Base> :type id
id :: a -> a
```

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

Tipuri supraîncărcate

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

```
Hugs.Base> :type (+)
(+) :: Num a => a -> a -> a
Hugs.Base> :type (<)
(<) :: Ord a => a -> a -> Bool
Hugs.Base> :type (/)
(/) :: Fractional a => a -> a -> a
```