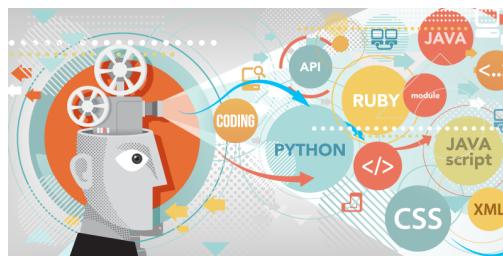


Llenguatges de Programació



Jordi Petit i Gerard Escudero

Contingut (1/2)

- **Presentació del curs**

Objectius. Continguts. Professorat. Avaluació. Bibliografia.

- **Conceptes bàsics**

Introducció. Característiques. Història. Ús dels LPs. Paradigmes. Turing completeness. Sistemes d'execució. Sistemes de tipus. Exercicis.

- **λ -càlcul**

Introducció. Estructura bàsica. Codificacions de Church. Recursivitat. Universalitat. Calculadores.

- **Programació funcional en Haskell**

Introducció. Funcions d'ordre superior. Llistes per comprensió. Avaluació mandrosa. Tipus algebraics. Mònades.

- **Raonament equacional**

Introducció. Base. Inducció. Inducció amb arbres. Millora eficiència. Sumari. Exercicis.

Contingut (2/2)

- **Introducció a la compilació**

Introducció. Visió general. Anàlisi lèxica. Anàlisi sintàctica. Arbres de sintaxi abstracta. Anàlisi semàntica. Interpretació.

- **Programació en Python**

Elements bàsics. Part funcional. Classes. Lazyness. Tipus algebraics. Compiladors.

- **Inferència de tipus**

Introducció. Algorisme de Milner. Exercicis.

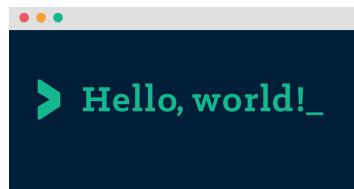
- **Conceptes avançats**

Recursivitat. Orientació a objectes. Subtipus i variància de tipus. Clausures. Concurrència.

- **Treball dirigit**

Llenguatges de Programació

Presentació del curs



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



1 / 8

Contingut

- Objectius
- Continguts
- Professorat
- Avaluació
- Bibliografia

2 / 8

Objectius

- Conèixer millor els llenguatges de programació
- Conèixer les característiques dels llenguatges funcionals
- Conèixer les característiques dels llenguatges de scripting
- Conèixer construccions avançades dels LPs
- Conèixer els elements bàsics de la compilació
- Millorar la capacitat d'aprendre nous llenguatges
- Millorar la capacitat de triar el LP adient

3 / 8

Continguts

- Fonaments de Llenguatges de Programació
- Llenguatges funcionals: Haskell
- Llenguatges de scripting: Python
- Introducció a la compilació
- Sistemes de tipus

4 / 8

Professorat



- Jordi Petit jordi.petit-silvestre@upc
- Gerard Escudero gerard.escudero@upc
- Edelmira Pasarella edelmira.pasarella@upc
- Jordi Delgado jordi.delgado@upc

5 / 8

Mètode d'avaluació

La **Guia docent** estableix aquesta avaluació: $N = 0.40 \cdot F + 0.25 \cdot P1 + 0.25 \cdot P2 + 0.10 \cdot D$, on:

- P1 = nota de l'examen parcial (mitjans de curs)
- P2 = nota de la pràctica (finals de curs)
- D = nota del treball dirigit
- F = nota de l'examen final

El parcial serà un examen a Jutge.org sobre programació funcional en Haskell.

La pràctica consisteix en utilitzar eines per generar compiladors i Python per a resoldre un cas pràctic.

El treball dirigit consisteix en preparar un vídeo i un document escrit sobre les propietats d'un llenguatge de programació. S'usarà avaluació entre companys (co-avaluació).

Les qualificacions de les competències transversals s'obtenen del treball dirigit.

Nota: Els repetidors poden reusar les seves notes del TD del curs passat.

El final serà un examen escrit que avaluarà sobre *tots* els continguts del curs.

6 / 8

Calendari

- Classes:
 - Inici: dv 9 de setembre
 - Final: dv 20 de desembre
- Exàmens:
 - Examen parcial: dj 7 de novembre (15:30 - 17:30)
 - Examen final: dj 9 de gener (15:00 - 18:00)
- Treballs:
 - Lliurament pràctica: dt 8 de gener (08:00)
 - Lliurament treball dirigit: dl 18 de desembre (08:00)
 - Correcció treball dirigit: dl 8 de gener (08:00)

7 / 8

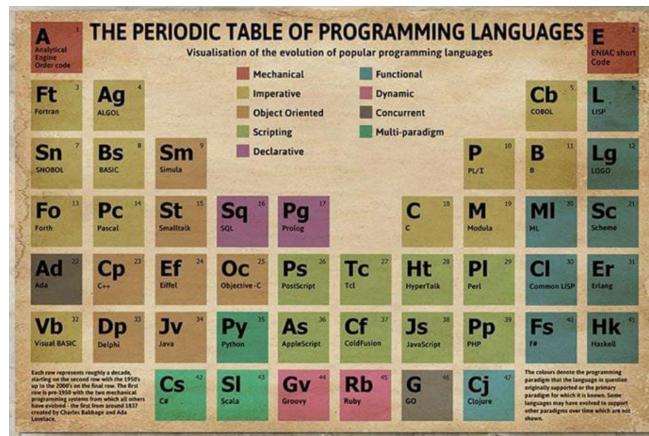
Bibliografia

- *Compiler design*. Wilhelm, R.; Maurer, D, Addison-Wesley, 1995.
- *Compilers: principles, techniques, and tools*. Aho, A.V.; Sethi, R.; Ullman, J.D, Addison-Wesley, 2007.
- *Concepts in programming languages*. Mitchell, J.C, Cambridge University Press, 2003.
- *Programming language pragmatics*. Scott, M.L, Morgan Kaufmann, 2009.
- *Haskell: the craft of functional programming*. Thompson, S, Addison-Wesley, 2011.
- *Razonando con Haskell: un curso sobre programación funcional*. Ruiz Jiménez, B.C, Thomson-Paraninfo, 2004.
- *Learn You Haskell for a Great Good*. Miran Lipova  a. [Disponible online](#).
- *Think Python*. Downey, B. O'Reilly, 2015. [Disponible online](#).

8 / 8

Llenguatges de Programació

Conceptes bàsics



Albert Rubio, Jordi Petit, Fernando Orejas, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

Introducció

Un llenguatge de programació (LP) és un llenguatge formal utilitzat per controlar el comportament d'un computador tot implementant un algorisme.



3 / 75

Introducció

Tradicionalment, els LPs es consideren des de tres angles (anàlegs a la lingüística):

- Sintaxi: la forma dels programes correctes.
 - Semàntica: el significat de les construccions dels programes.
 - Pragmàtica: com s'usa el LP.

La majoria d'LPs venen definits per

- una especificació estàndard
ex: The C Programming Language, 1978
 - una implementació de referència
ex: Standard ML

Perquè estudiar els LPs?

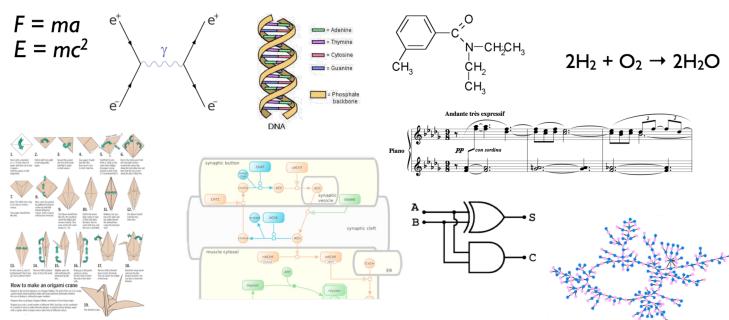
- Visió enginyeril: Necessitem LPs per programar computadors per fer coses xules.
- Visió científica: Necessitem LPs per descriure i raonar sobre aquestes computacions.

5 / 75

Perquè estudiar els LPs?

Els LPs, en tant que llenguatges, ens permeten representar conceptes i són una eina de pensament.

Exemples:



6 / 75

Perquè estudiar els LPs?

Els llenguatges que usem:

- influencien les nostres percepcions,
- guien i donen suport al nostre raonament,
- permeten i faciliten la nostra comunicació.

Exemple: Representacions dels nombres:

$$II < \overline{W} \ll II / < \overline{W} W = \overline{W}$$

$$\overline{VIII}CXCII \div MXXIV = VIII$$

$$8192 / 1024 = 8$$

$$2^{13} / 2^{10} = 2^{13 - 10} = 2^3$$

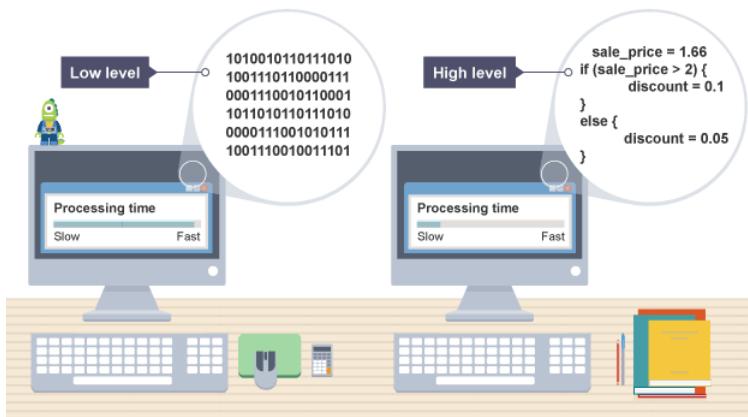
7 / 75

Perquè estudiar els LPs?

Els llenguatges que usem:

- influencien les nostres percepcions,
- guien i donen suport al nostre raonament,
- permeten i faciliten la nostra comunicació.

Exemple: Codi màquina *vs* codi alt nivell:



8 / 75

Perquè estudiar els LPs?

Els llenguatges que usem:

- influencien les nostres percepcions,
- guien i donen suport al nostre raonament,
- permeten i faciliten la nostra comunicació.

Exemple: Programació imperativa *vs* declarativa:

```
let xs = [1, 2, 3, 4, 5]
let ys = []
for (var i = 0; i < xs.length; i++) {
    ys.push(xs[i] * 2);
}
```

```
let xs = [1, 2, 3, 4, 5]
let ys = xs.map(function (x,i) {
    return 2*x
})
```

9 / 75

Com estudiar els LPs?

Estudiar-los tots.



Estudiar les seves característiques, els seus usos, avaluar les seves qualitats, classificar-los en famílies, conèixer la seva història, estudiar els seus fonaments...



10 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

11 / 75

Característiques bàsiques d'un LP

- Tipus de dades: amb quins dades i objectes treballem.
- Sistema de tipus.
- Control de seqüència: en quin ordre s'executen les operacions.
- Control de dades. Com s'accedeix a les dades i als objectes.
- Entrada / Sortida.

12 / 75

Qualitats d'un LP

- Llegibilitat
- Eficiència
- Fiabilitat
- Expressivitat
- Simplicitat
- Nivell d'abstracció
- Adequació als problemes a tractar
- Facilitat d'ús
- Ortogonalitat

13 / 75

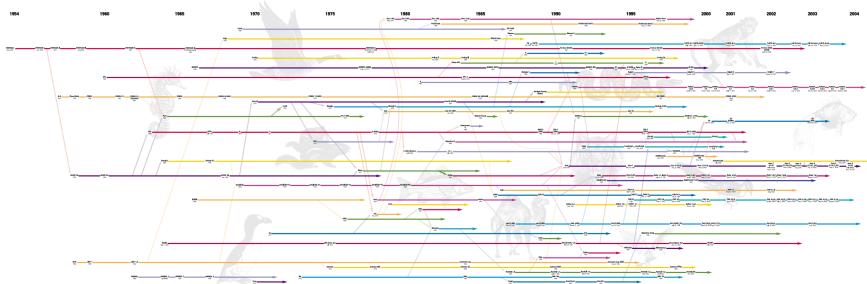
Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

14 / 75

Història

📘 *History of Programming Languages*, O'Reilly



15 / 75

Història: Orígens

Tauletes babilòniques (2000ac)

A [rectangular] cistern.

The height is 3, 20, and a volume of 27, 46, 40 has been excavated. The length exceeds the width by 50. You should take the reciprocal of the height, 3, 20, obtaining 18. Multiply this by the volume, 27, 46, 40, obtaining 8, 20. Take half of 50 and square it, obtaining 10, 25. Add 8, 20, and you get 8, 30, 25. The square root is 2, 55. Make two copies of this, adding [25] to the one and subtracting from the other. You find that 3, 20 [i.e., 3 1/3] is the length and 2, 30 [i.e., 2 1/2] is the width.

This is the procedure.



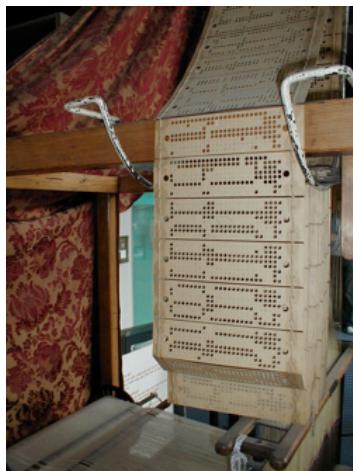
📘 *Ancient Babylonian Algorithms*. D.E. Knuth, Communications ACM 1972.

🎥 *Math whizzes of ancient Babylon figured out forerunner of calculus*, YouTube.

16 / 75

Història: Orígens

Teler de Jacquard (1804)



Fotos: <http://www.revolutionfabrics.com/blog/2018/9/26/the-jacquard-loom-and-the-binary-code>

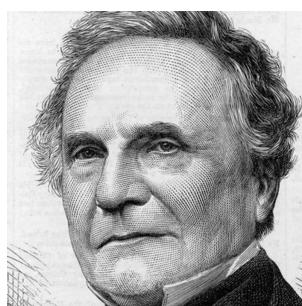
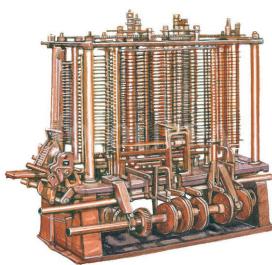
🎬 How an 1803 Jacquard Loom Lead to Computer Technology

17 / 75

Història: Orígens

Màquina analítica de Charles Babbage (1842)

Es considera que Ada Lovelace és la primera programadora.

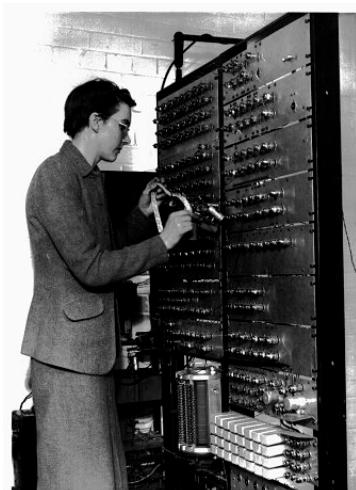


Fotos: Domini públic

🎬 Ada Lovelace, The World's First Computer Nerd?

18 / 75

Història: Ensambladors



Kathleen Booth va escriure el primer llenguatge ensamblador (per un ordinador ARC al 1947).

11)	$M \times R \rightarrow cA$	Clear accumulator, multiply N by R and place L.H. 39 digits of answer in A and R.E. 39 digits in R.
12)	$A \div M \rightarrow cR$	Clear register, divide A by M, leave quotient in R and remainder in A.
13)	$C \rightarrow M_1$	
14)	$C \rightarrow M_2$	
15)	$C_0 \rightarrow M_1$	If number in A > 0 shift control to M_1 .
16)	$C_0 \rightarrow M_2$	
17)	$A \rightarrow M$	

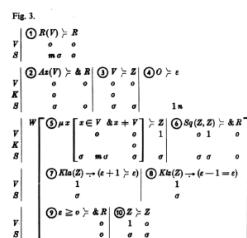
Fotos: Domini públic

🕒 Kathleen Booth, Assembling Early Computers While Inventing Assembly

19 / 75

Història: Plankalkül

Primer LP d'alt nivell dissenyat per Konrad Zuse (1942-1945) pel seu ordinador a relés Z4. Implementat al 1990.



Fotos: Domini públic

20 / 75

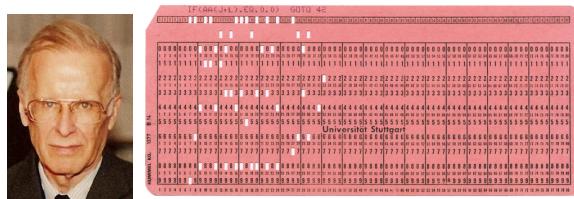
Història: Fortran

FORmula TRANslator (1954-1957). Desenvolupat per John Bakus a IBM per a computació científica.

Es volia generar codi comparable al programat en ensamblador.

Idees principals:

- Variables amb noms (6 caràcters)
- Bucles i condicionals aritmètics
- E/S amb format
- Subrutines
- Taules



Fotos: Domini públic i Wikipedia

21 / 75

Història: Fortran

```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
READ INPUT TAPE 5, 501, IA, IB, IC
501 FORMAT (I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE OR ZERO
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C MUST BE GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
    IF (IA) 777, 777, 701
701 IF (IB) 777, 777, 702
702 IF (IC) 777, 777, 703
703 IF (IA+IB-IC) 777, 777, 704
704 IF (IA+IC-IB) 777, 777, 705
705 IF (IB+IC-IA) 777, 777, 799
777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
799 S = FLOATF (IA + IB + IC) / 2.0
        AREA = SQRTF( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
        + (S - FLOATF(IC)))
        WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601 FORMAT (4H A= ,I5,5H B= ,I5,5H C= ,I5,8H AREA= ,F10.2,
        + 13H SQUARE UNITS)
        STOP
        END
```

22 / 75

Història: Fortran

THE FIRST PROGRAM

The first error message

The running of the first program, though well documented by Herb Bright [15], was not a planned activity. There is one error in Bright's report which needs correction -- 1957 April 20 was a Saturday not a Friday as reported. It was on the afternoon of that Friday when an unmarked deck of cards was delivered to Westinghouse-Bettis and which was assumed by Lew Ondis to be the right size to be a FORTRAN compiler. Jim Callaghan quickly wrote a small program based on a recent technical report by Ollie Swift, and using the "common" technique for running programs on the IBM 704, the unmarked deck was loaded into the system followed by the program. Surprisingly it worked and in a short time the first FORTRAN error message was output:

FORTRAN DIAGNOSTIC PROGRAM RESULTS

```
05065 SOURCE PROGRAM ERROR. THIS IS A TYPE-GO TO ( ),I  
BUT THE RIGHT PARENTHESIS IS NOT FOLLOWED BY A COMMA
```

END OF DIAGNOSTIC PROGRAM RESULTS

The error was quickly fixed and the program (apparently) recompiled and executed to produce "... a whiff of computing followed by 28 pages of output ..."

Font: J.A.N. Lee, Twenty Five Years of Fortran

23 / 75

Història: COBOL

Common Business Oriented Language (1959). Desenvolupat per Grace Hopper pel DoD i fabricants per a aplicacions de gestió.

Idees principals:

- Vol semblar idioma anglès, sense símbols
- Macros
- Registres
- Fitxers
- Identificadors llargs (30 caràcters)



Foto: Domini públic

24 / 75

Història: COBOL

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. Multiplier.  
AUTHOR. Michael Coughlan.  
* Example program using ACCEPT, DISPLAY and MULTIPLY to  
* get two single digit numbers from the user and multiply them together  
  
DATA DIVISION.  
  
WORKING-STORAGE SECTION.  
01 Num1          PIC 9  VALUE ZEROS.  
01 Num2          PIC 9  VALUE ZEROS.  
01 Result        PIC 99 VALUE ZEROS.  
  
PROCEDURE DIVISION.  
    DISPLAY "Enter first number (1 digit) : " WITH NO ADVANCING.  
    ACCEPT Num1.  
    DISPLAY "Enter second number (1 digit) : " WITH NO ADVANCING.  
    ACCEPT Num2.  
    MULTIPLY Num1 BY Num2 GIVING Result.  
    DISPLAY "Result is = ", Result.  
    STOP RUN.
```

25 / 75

Història: LISP

LISP Processing (1958). Desenvolupat per John McCarthy al MIT per a recerca en IA.

Idees principals:

- Sintaxi uniforme
- Funcions (composició i recursivitat)
- Llistes
- Expressions simbòliques
- Recol·lector de brossa



Foto: Wikipedia

26 / 75

Història: LISP

```
(defun factorial (N)
  "Compute the factorial of N."
  (if (= N 1)
    1
    (* N (factorial (- N 1)))))

(defun first-name (name)
  "Select the first name from a name represented as a list."
  (first name))

(setf names '((John Q Public) (Malcolm X)
              (Admiral Grace Murray Hopper) (Spot)
              (Aristotle) (A A Milne) (Z Z Top)
              (Sir Larry Olivier) (Miss Scarlet)))
```

27 / 75

Història: Algol

ALGOrithmic Language (1958). Dissenyat com un llenguatge universal per computació científica. No gaire popular, però dóna lloc a LPs com Pascal, C, C+, and Java.

Idees principals:

- Blocs amb àmbits de variables
- Pas per valor i pas per nom (\neq pas per referència)
- Recursivitat
- Gramàtica formal (Backus-Naur Form or BNF)

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
  value n, m; array a; integer n, m, i, k; real y;
begin
  integer p, q;
  y := 0; i := k := 1;
  for p := 1 step 1 until n do
    for q := 1 step 1 until m do
      if abs(a[p, q]) > y then
        begin y := abs(a[p, q]);
          i := p; k := q
        end
  end Absmax
```

28 / 75

Història: Algol

Pas per nom:

```
begin
    integer i;
    integer array A[0:9];

    procedure ini (x); integer x; (* pas per nom *)
    begin
        for i := 0 step 1 until 9 do x := 0
    end;

    ini(A[i]);
    ...
end
```

El paràmetre `x` es passa per nom (amb `value x`; es passaria per valor).

Dins de `ini()`, aparentment, s'assigna 10 cops 0 a `x`.

Però `x` és el "nom" del paràmetre real, és a dir `A[i]`.

Per tant, `ini()` realment inicialitza a zero tot l'array.

29 / 75

Exercici

Què fa aquest programa?

```
begin
    integer i, sum;
    integer array A[0:9];

    procedure suma (x, s); integer x, s;
    begin
        s := 0;
        for i := 0 step 1 until 9 do s := s + x
    end;

    suma(A[i], sum);
end
```

30 / 75

Història: altres llenguatges

- Basic (Orientat a l'ensenyament de la programació) - 1964
- Pascal/Algol 68 (els hereus directes d'Algol 60) - 1970/1968
- C (Definit per programar Unix) - 1972
- Prolog (Primer llenguatge de programació lògica) - 1972
- Simula 67/Smalltalk 80 (primers llenguatges OO)
- Ada (LP creat per ser utilitzat pel Departament de Defensa Americà) - 1980
- ML/Miranda (primers llenguatges funcionals moderns) - 1983/1986
- C++ (llenguatge dinàmic i flexible compatible amb C) - 1983
- Java (llenguatge orientat a objectes per a torradores) - 1990
- Python (llenguatge llegible d'alt nivell de propòsit general) - 1991
- ...

31 / 75

Contingut

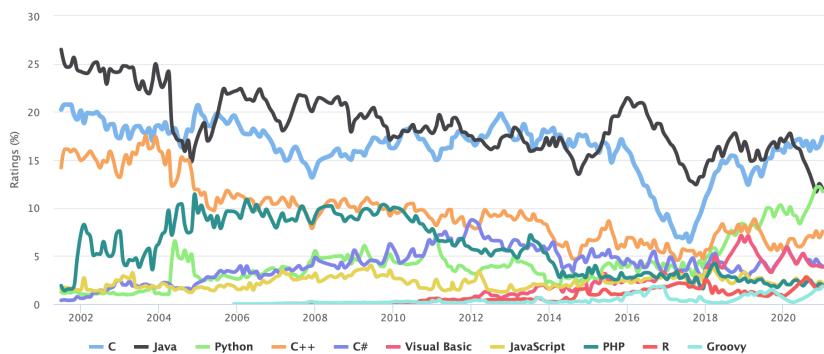
- Introducció
- Característiques
- Història
- [Ús dels LPs](#)
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

32 / 75

Ús dels LPs

Com mesurar la popularitat dels LPs?

- TIOBE Programming Community Index



Font: <https://www.tiobe.com/tiobe-index/> (2021)

33 / 75

Ús dels LPs

Com mesurar la popularitat dels LPs?

- IEEE Spectrum ranking: The Top Programming Languages 2018

Language Rank	Types	Spectrum Ranking
1. Python	🌐💻	100.0
2. C	📱💻	99.7
3. Java	🌐📱💻	99.5
4. C++	📱💻	97.1
5. C#	🌐📱💻	87.7
6. R	💻	87.7
7. JavaScript	🌐📱	85.6
8. PHP	🌐	81.2
9. Go	🌐💻	75.1
10. Swift	📱💻	73.7

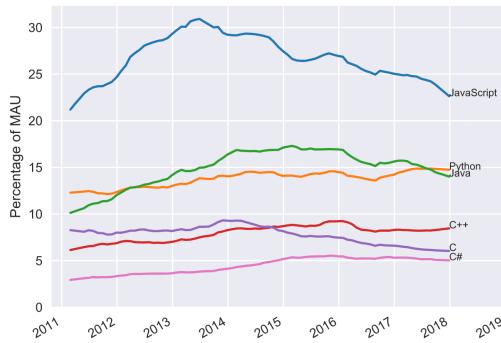
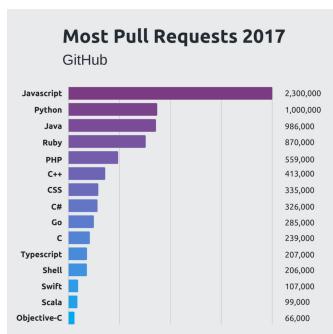
Font: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018> (2018)

34 / 75

Ús dels LPs

Com mesurar la popularitat dels LPs?

- Estadístiques de GitHub

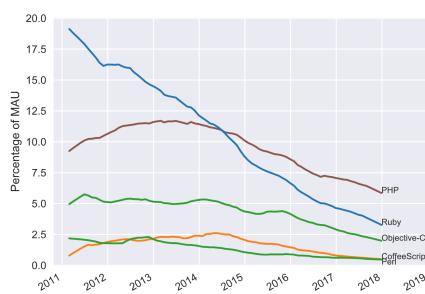
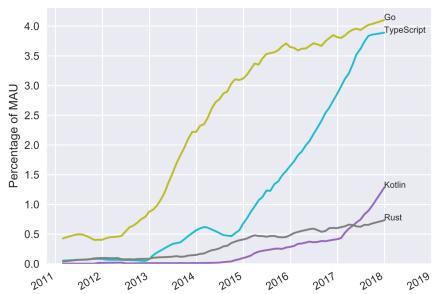


Fonts: <https://github.com> i <https://www.benfrederickson.com/ranking-programming-languages-by-github-users/>

35 / 75

Ús dels LPs

Quins LPs estudiar/evitar?

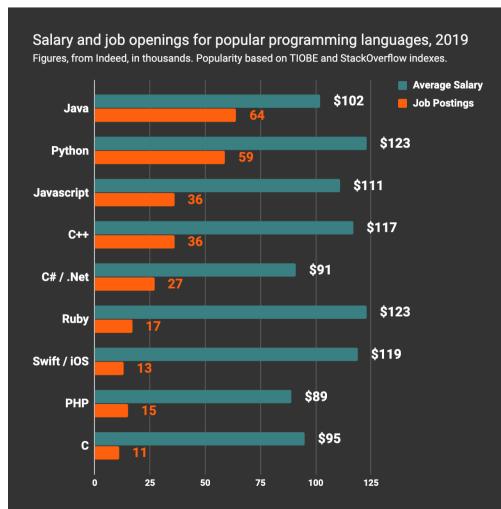


Font: <https://www.benfrederickson.com/ranking-programming-languages-by-github-users/>

36 / 75

Ús dels LPs

Quins LPs estudiar/evitar?



Font: <https://www.codeplatoon.org/the-best-paying-and-most-in-demand-programming-languages-in-2019/>

37 / 75

Ús dels LPs (1965 - 2019)

Most Popular Programming Languages 1965 - 2019



38 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

39 / 75

Paradigmes de LPs

Els paradigmes de programació classifiquen els LPs segons les seves característiques.

The Paradigms of Programming

Robert W. Floyd
Stanford University



Today I want to talk about the paradigms of programming, how they affect our success as developers of computer programs, how they should be taught, and how they should be embodied in our programming languages.

A familiar example of a paradigm of programming is the technique of *structured programming*, which appears to be the dominant paradigm in most current treatments of programming methodology. Structured programming, as formalized by Dijkstra [6], Wirth [27, 28], and Parnas [21], among others, consists of two phases.

In the first phase, that of top-down design, or stepwise refinement, the problem is decomposed into a very small number of smaller subproblems. Using Gaussian elimination of simultaneous linear equations, say, the first level of decomposition would be into a stage of triangularizing the equations and a following stage of back-substitution in the triangulated system. This gradual decomposition is continued until the subproblems that arise are simple enough to completely solve. In the simultaneous equation example, the back substitution process would be further decomposed as a backwards iteration of a process which finds and stores the value of the i th variable from the i th equation. Yet further decomposition would yield a fully detailed algorithm.

In the second phase, that of bottom-up construction, the paradigm entails working upward from the concrete objects and functions of the underlying machine to the more abstract objects and functions used throughout the modules produced by the top-down design. In the linear equation example, if the coefficients of the equations are rational functions of one variable, we might first design

Paradigm(пе́радім) —, [а. ф. *paradigma*, ад. л. *paradigma*, а. гр. παράδειγμα pattern, example, f. παράδειξ-ναι to exhibit beside, show side by side. . .]
1. A pattern, exemplar, example.

1752 J. Gill *Trinity v. 91*

The archetype, pattern, exemplar, and idea, according to which all things were made.

From the Oxford English Dictionary.

Permission to copy without fee all or part of this material is granted provided that: 1. copies are not made or distributed for commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc.; 2. the full bibliographic reference is given; otherwise, or in republish, requires a fee and/or specific permission. Write, addres: Department of Computer Science, Stanford University, Stanford, CA 94305.
© 1979 ACM 0001-0732/79/0800-0455 \$00.75.

455

Communications
of
the ACM
August 1979
Volume 22
Number 8

Paradigmes de LPs

Paradigmes comuns:

- Imperatiu: Les instruccions precisen els canvis d'estat.

```
f = 1;  
while (--n) f *= n;
```

- Declaratiu: Es caracteritza el resultat, però no com calcular-lo.

```
select full_name, order_date, order_amount  
from customers inner join orders  
on customers.customer_id = orders.customer_id
```

- Funcional: El resultat és el valor d'una sèrie d'aplicacions de funcions.

```
cincMesGrans :: [a] -> [a]  
cincMesGrans = take 5 . reverse . sort
```

41 / 75

Paradigma imperatiu

Característiques:

- Noció d'estat
- Instruccions per canviar l'estat
- Efectes laterals

Exemples:

- C/C++, Python, Java, Ensamblador, ...

Útils quan, per exemple, l'eficiència és clau.

42 / 75

Paradigma imperatiu

Subclassificacions:

- Procedural: Les instruccions s'agrupen en procediments.

```
PROCEDURE swap (VAR a, b: INTEGER)
  VAR c: INTEGER;
  BEGIN
    c := a;  b := a;  a := c;
  END;
```

- Orientat a objectes: Les instruccions s'agrupen amb l'estat dels objectes sobre les quals operen.

```
Point»dist: aPoint
  dx := aPoint x - x.
  dy := aPoint y - y.
  ↑ ((dx * dx) + (dy * dy)) sqrt
```

43 / 75

Paradigma declaratiu

Característiques:

- Llenguatges descriptius.
- El programa diu què s'ha de fer, però no necessàriament com.

Utilitat:

- Prototipat d'aplicacions amb forta component simbòlica, problemes combinatoris, etc.
- Consultes en bases de dades relacionals o lògiques.
- Per especificació i raonament automàtic.

Exemples:

- SQL (consultes relacionals) / GraphQL (consultes en grafs, amb tipus) / Prolog (lògica de primer ordre) / Matemàtiques

44 / 75

Paradigma declaratiu

SQL: Trobar tots els emails dels usuaris del Jutge amb el Hello World acceptat.

```
EXPLAIN
SELECT DISTINCT email
FROM Users JOIN Submissions USING (user_id)
WHERE problem_id='P68688_en' AND veredict='AC'
ORDER BY email
```

Planificador (postgres):

```
QUERY PLAN
-----
Unique  (cost=56242.20..56290.03 rows=9566 width=30)
 -> Sort  (cost=56242.20..56266.11 rows=9566 width=30)
     Sort Key: users.email
     -> Hash Join  (cost=2889.46..55609.71 rows=9566 width=30)
         Hash Cond: (submissions.user_id = users.user_id)
         -> Bitmap Heap Scan on submissions  (cost=570.63..53265.77 rows=95
             Recheck Cond: (problem_id = 'P68688_en'::text)
             Filter: (veredict = 'AC'::text)
             -> Bitmap Index Scan on idx_submissions_problem_id  (cost=0.1
                 Index Cond: (problem_id = 'P68688_en'::text)
         -> Hash  (cost=1860.59..1860.59 rows=36659 width=37)
             -> Seq Scan on users  (cost=0.00..1860.59 rows=36659 width=3
```

45 / 75

Paradigma declaratiu

Subclasificacions:

- Consultes: Resposta a consultes a una base de dades.

```
SELECT full_name, order_date, order_amount
FROM customers INNER JOIN orders
ON customers.customer_id = orders.customer_id
```

- Matemàtic: El resultat es declara com a la solució d'un problema d'optimització.

```
Maximize
  x1 + 2 x2 + 3 x3 + x4
Subject To
  - x1 + x2 + x3 + 10 x4 ≤ 20
  x1 - 3 x2 + x3 ≤ 30
  x2 - 3.5 x4 = 0
```

- Lògic: Resposta a una pregunta amb fets i regles.

```
human(socrates).
mortal(X) :- human(X).
?- mortal(socrates).
```

46 / 75

Paradigma funcional

Característiques:

- Procedural
- Sense noció d'estat i sense efectes laterals
- Més fàcil de raonar (sobre correctesa o sobre transformacions)

Utilitat:

- Útils per al prototipat, fases inicials de desenvolupament (especificacions executables i transformables).
- Tractament simbòlic.
- Sistemes de tipus potents (incloent polimorfisme paramètric i inferència de tipus).

Exemples: Haskell, ML (Caml, OCaml), Erlang, XSLT (tractament XML),...

47 / 75

Paradigma funcional

Conceptes clau:

- Recursivitat:

```
(define (fib n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (else
      (+ (fib (- n 1))
          (fib (- n 2))))))
```

- Funcions d'ordre superior: funcions que reben o retornen funcions.

```
map(lambda x: 2*x, [1,2,3,4])
```

- Aplicació parcial (currying):

```
doble = (*2)
```

48 / 75

Paradigma funcional

Conceptes clau:

- Funcions pures: Amb els mateixos arguments, les funcions sempre retornen el mateix resultat. No hi ha efectes laterals.
- Mecanismes d'avaluació: Avaluació estricta *vs* avaluació mandrosa.
- Sistemes de tipus.

49 / 75

Paradigma OO

Característiques:

- Es basa en *objectes* (atributs + mètodes) i potser *classes*.
- Inclou principalment *polimorfisme* (*subtipat*) i *herència*.
- Poden tenir sistemes de tipus complexos.

Exemples: Smalltalk, Simula, C++, Java...

50 / 75

Llenguatges multiparadigma

Molts LPs combinen diferents paradigms. Per exemple:

- Python, Perl: imperatiu + orientat a objectes + funcional
- OCaml: funcional + imperatiu + orientat a objectes

Alguns LPs incorporen característiques d'altres paradigmes:

- Prolog: lògic (+ imperatiu + funcional)

Altres combinacions:

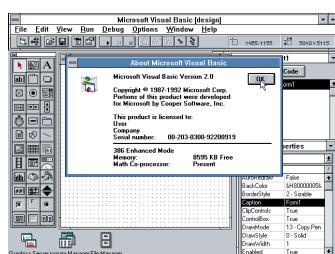
- Erlang: funcional + concurrent + distribuït

51 / 75

Llenguatges visuals

En un llenguatges de programació visual (VPL) els programes són creats manipulant elements gràfics.

- Visual Basic: Dialecte de BASIC utilitzant una interfície gràfica per facilitar la creació d'interfícies gràfiques (1991).



- Scratch: Una d'educació per a nens a partir de 8 anys (2003).



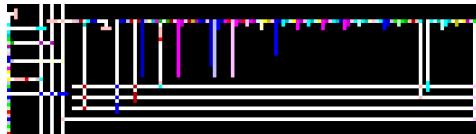
52 / 75

Llenguatges esotèrics

- Brainfuck: Basat en màquines de Turing, només té 8 instruccions. Hello world:

```
,>+++++[<----->-],[<+>-],<.>.
```

- Piet: Usa imatges amb 20 colors com a codi font. Factorial:



- Whitespace: Només té espais en blanc i tabuladors. BFS:

(indenteu-lo bé quan el copieu 😊)

53 / 75

Llenguatges esotèrics

- Shakespeare: Amaga un programa dins d'una obra de teatre.

The Infamous Hello World Program.

Romeo, a young man with a remarkable patience.
Juliet, a likewise young woman of remarkable grace.
Ophelia, a remarkable woman much in dispute with Hamlet.
Hamlet, the flatterer of Andersen Insulting A/S.

Act I: Hamlet's insults and flattery.

Scene I: The insulting of Romeo.

[Enter Hamlet and Romeo]

Hamlet:

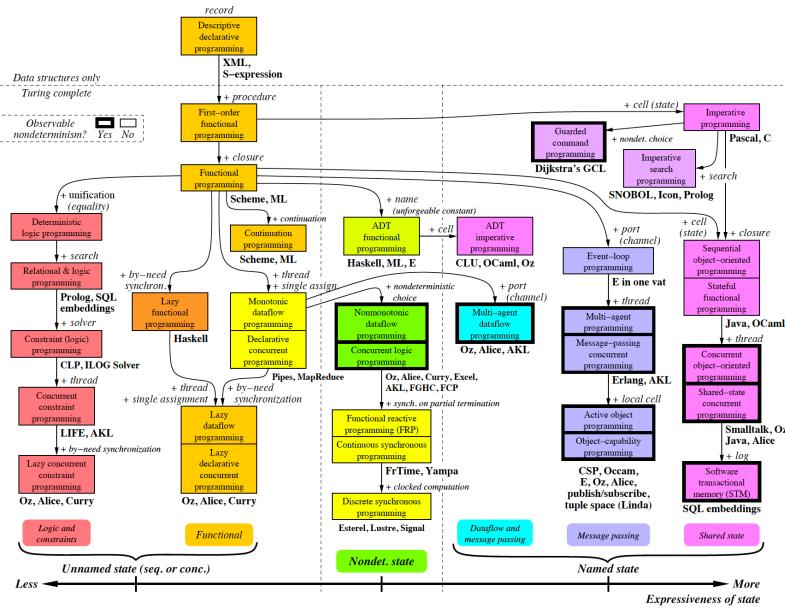
You lying stupid fatherless big smelly half-witted coward!
You are as stupid as the difference between a handsome rich brave
hero and thyself! Speak your mind!

You are as brave as the sum of your fat little stuffed misused dusty
old rotten codpiece and a beautiful fair warm peaceful sunny summer's
day. You are as healthy as the difference between the sum of the
sweetest reddest rose and my father and yourself! Speak your mind!

You are as cowardly as the sum of yourself and the difference
between a big mighty proud kingdom and a horse. Speak your mind.

54 / 75

Paradigmes de LPs: Més taxonomies



Font: *Programming Paradigms for Dummies*, P. Van Roy

55 / 75

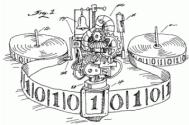
Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

56 / 75

Turing completeness

Màquina de Turing: model matemàtic de càlcul imperatiu molt simple
(Alan Turing, 1936)



- Cinta infinita amb un capçal mòbil per llegir/escriure símbols
- Conjunt finit d'estats
- Funció de transició (estat, símbol → estat, símbol, moviment)

λ -càlcül: model matemàtic de càlcul funcional molt simple.
(Alonzo Church, 1936).



$$\begin{array}{c} (\lambda \textcolor{red}{y} x(\textcolor{red}{yz})) \textcolor{blue}{(ab)} \\ \downarrow \\ x(\textcolor{blue}{abz}) \end{array}$$

- Sistema de reescritura
- Basat en abstracció i aplicació de funcions

Tesi de Church-Turing: "Tot algorisme és computable amb una Màquina de Turing o amb una funció en λ -càlcül".

Fotos: Wikipedia, Fair Use, [Lambda Calculus for Absolute Dummies](#)

57 / 75

Turing completeness

Un LP és Turing complet si pot implementar qualsevol càlcul que un computador digital pugui realitzar.

Alguns autors consideren només com a LPs els llenguatges Turing complets.

- LPs Turing complets:
 - LPs de programació de propòsit general (C/C++, Python, Haskell...)
 - automàts cel·lulars (Joc de la vida, ...)
 - alguns jocs (Minecraft, Buscamines...)

Per ser Turing complet només cal tenir salts condicionals (bàsicament, `if` i `goto`) i memòria arbitràriament gran.

- LPs no Turing complets:
 - Expressions regulars (a Perl o a AWK)
 - ANSI SQL

58 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completesa
- **Sistemes d'execució**
- Sistemes de tipus
- Exercicis

59 / 75

Sistemes d'execució

- Compilat: el codi és transforma en codi objecte i després es monta en un executable. Sol ser eficient. Es distribueix l'executable.

Exemples: C, C++, Ada, Haskell, ...

- Interpretat: el codi s'executa directament o el codi es transforma en codi d'una màquina virtual, que l'executa. Es distribueix el codi font.

Aumenten la portabilitat i l'expressibilitat (es poden fer més coses en temps d'execució) però disminueix l'eficiència.

Exemples: Python, JavaScript, Prolog (WAM), Java (JVM), ...

Sistemes mixtes:

- Just in Time compilation: Es compila (parcialment) en temps d'execució.
- Alguns interpretats, poden ser també compilats (per exemple, Prolog).
- I al revés (Haskell).

⇒ El sistema d'execució depèn més de la implementació que del LP.

60 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- **Sistemes de tipus**
- Exercicis

61 / 75

Sistemes de tipus

Un sistema de tipus és un conjunt de regles que assignen *tipus* als elements d'un programa (com ara variables, expressions, funcions...) per evitar errors.

La comprovació de tipus verifica que les diferents parts d'un programa es comuniquin adequadament en funció dels seus tipus.

Per exemple, amb:

```
class Persona;  
class Forma;  
class Rectangle :Forma;  
class Triangle :Forma;  
  
double area (const Forma&);
```

- Cridar a `area` amb un `Rectangle` o un `Triangle` és correcte,
- Cridar a `area` amb una `Persona` o un enter és un error de tipus.

62 / 75

Errors de tipus

Un error de tipus consisteix en aplicar a les dades una operació que el seu tipus no suporta.

- Type Cast: usar un valor d'un tipus en un altre tipus.
 - 👉 En C, un enter pot ser usat com a funció (com a adreça), però pot saltar on no hi ha una funció i pot provocar un error.
 - 👉 En C, `printf("%s", 42);`.
- Aritmètica de punters.
 - 👉 En C++, si tenim `A p[10];` llavors `p[15]` té tipus A, però el que hi ha a `p[15]` pot ser qualsevol altra cosa i pot provocar un error de tipus.
- Alliberament explícit de memòria (deallocate/delete).

Exemple: En Pascal usar un apuntador alliberat pot donar errors de tipus.
- En llenguatges OO (antics), no existència d'un mètode (degut a l'herència).

63 / 75

Seguretat de tipus

La seguretat de tipus (*type safety*) és la mesura de com de fàcil/difícil és cometre errors de tipus en un LP.

Exemples:

- C té reputació de ser un LP sense gaire seguretat de tipus (unions, enumerats, `void*`, ...).
- C++ hereta C però proporciona més seguretat de tipus: els enumerats no es poden converteixen implícitament amb els enters o altres enumerats, el *dynamic casting* pot comprovar errors de tipus en temps execució, ...
- Java és dissenyat per a proporcionar seguretat de tipus. Però es pot abusar del *classloader*.
- Python, igual que Java, està dissenyat per donar seguretat de tipus, malgrat que el tipatge sigui dinàmic.
- Es creu que Haskell és *type safe* si no s'abusa d'algunes construccions com `unsafePerformIO`.

64 / 75

Tipat fort vs feble

Tipat fort (*strong typing*): L'LP imposa restriccions que eviten barrejar valors de diferents tipus (conversions explícites).

Tipat feble (*weak typing*): L'LP té regles màgiques per convertir tipus automàticament.

Javascript

```
4 + '7';      // '47'  
4 * '7';      // '28'
```

PHP

```
4 + '7';      // '11'  
4 * '7';      // '28'
```

Python

```
4 + '7'       # ✗ TypeError: unsupported operand type(s) for +: 'int' and 'str'  
4 * '7'       # '7777'
```

65 / 75



ShadowCheetah @shadowcheets · 12 d'ag. de 2019 · ...

Javascript is weird.

```
> ('b' + 'a' + + 'a' + 'a').toLowerCase()  
< "banana"
```

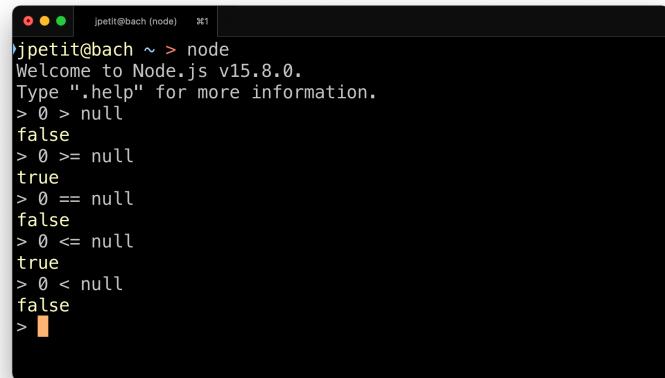
95

3m

6,6m

↑

66 / 75



```
jpettit@bach ~ > node
Welcome to Node.js v15.8.0.
Type ".help" for more information.
> 0 > null
false
> 0 >= null
true
> 0 == null
false
> 0 <= null
true
> 0 < null
false
> 
```

67 / 75

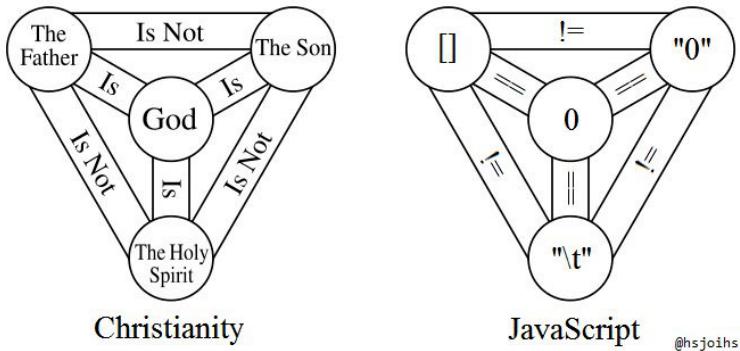
Arrays equal Strings?!

04 May 2018

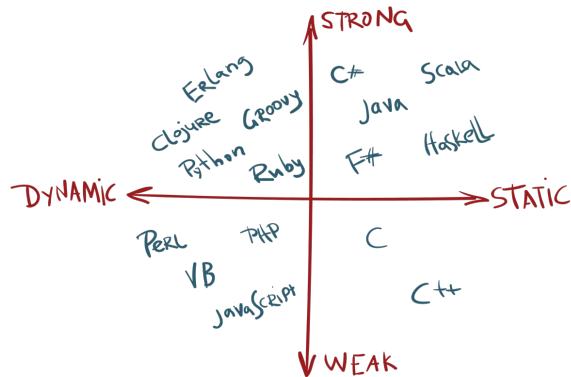
```
var a = [1,2,3];
var b = [1,2,3];
var c = '1,2,3';

a == c; // true
b == c; // true
a == b; // false
```

68 / 75



Sistemes de tipus



Font

71 / 75

Contingut

- Introducció
- Característiques
- Història
- Ús dels LPs
- Paradigmes
- Turing completeness
- Sistemes d'execució
- Sistemes de tipus
- Exercicis

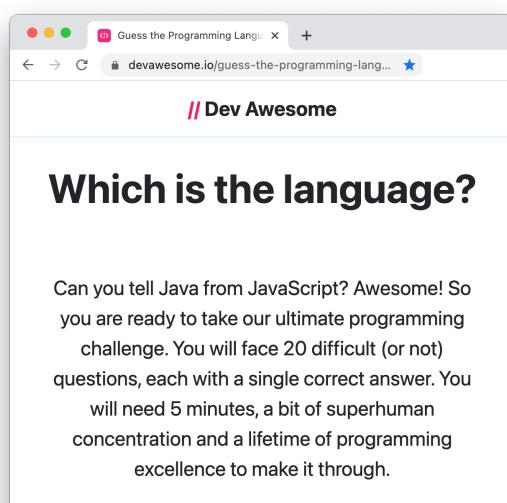
72 / 75

Exercicis

1. Digueu quin és el vostre LP favorit.
2. Expliqueu quines són les seves propietats fonamentals:
 - Propòsit del llenguatge.
 - Paradigma o paradigmes de programació que admet el llenguatge.
 - Sistema d'execució.
 - Sistema de tipus.
 - Principals aplicacions.
 - Història del LP i relació amb LPs similars.
 - Exemples de codi il·lustratius.
 - Altres característiques particulars.

73 / 75

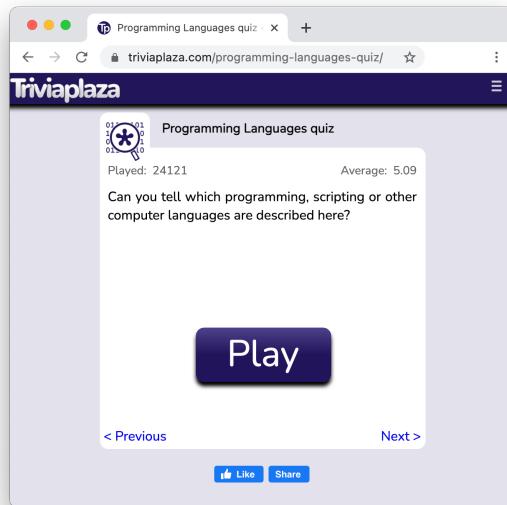
Exercicis



<https://devawesome.io/guess-the-programming-language/>

74 / 75

Exercicis

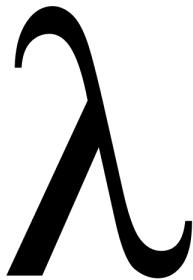


<https://www.triviaplaza.com/programming-languages-quiz/>

75 / 75

Llenguatges de Programació

lambda càcul



Albert Rubio, Jordi Petit, Fernando Orejas, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

Introducció

El λ -càlcul és un model de computació funcional, l'origen dels llenguatges funcionals, i la base de la seva implementació.

Inventat per Alonzo Church, cap al 1930.



AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.
By ALONZO CHURCH

1. **Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function $f(x)$ such that if x is a positive integer, $f(x) = 1$ is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving x .
Another example of a problem of this class is the following: to find a means of determining of any given positive integer n whether or not there exist positive integers x and y such that $x^y = n$. In this case it is required to find an effectively calculable function $f(x)$ such that $f(x) = 1$ if and only if there exist positive integers x and y such that $x^y = n$. Clearly the condition that the function f be effectively calculable is an essential part of the definition of the class of problems under consideration.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional manifolds. This problem can be interpreted as a problem of elementary number theory in view of the fact that it is required to find an effectively calculable function $f(x)$ such that $f(x) = 1$ if and only if x is a complete set of invariants of closed three-dimensional manifolds. In fact, as is well known, the property of a set of invariants that it appears as a complete set of invariants of closed three-dimensional manifolds is one of invariants that they represent homeomorphic complex, can both be expressed in purely number-theoretic terms. If we consider, in a straightforward manner, the problem of finding a complete set of invariants of closed three-dimensional manifolds, it will then be immediately possible that the problem reduces to the problem of finding an effectively calculable function $f(x)$ such that $f(x) = 1$ if and only if the whole set of invariants of closed three-dimensional manifolds is equivalent to the problem of finding an effectively calculable function $f(x)$ such that $f(x) = 1$ if and only if the whole set of invariants of closed three-dimensional manifolds is equivalent to x .

*Presented to the American Mathematical Society, April 10, 1930.

†The solution of the particular positive integer Σ instead of some other is, of course, accidental and immaterial.

Consisteix en agafar una línia de símbols i aplicar una operació de *cut-and-paste*.

$$(\lambda(y)x(yz)) (ab) \downarrow \\ x(abz)$$

Fotos: Fair Use, jstor.org, Lambda Calculus for Absolute Dummies

3 / 39

Contingut

- Introducció
- Estructura bàsica
 - Components
 - Avaluació
 - Macros
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

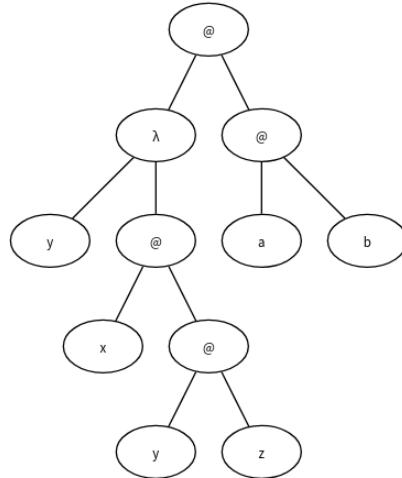
Gramàtica

```
terme → lletra | ( terme ) | abstracció | aplicació  
abstracció → λ lletra . terme  
aplicació → terme terme
```

Exemples de termes:

- \$x\$
- $\lambda x . x$
- $(\lambda y . x(yz)) (ab)$

Arbre de $(\lambda y . x(yz)) (ab)$:



5 / 39

Gramàtica

Les lletres es diuen *variables* i no tenen cap significat. El seu nom no importa. Si dues variables tenen el mateix nom, són la mateixa cosa.

Els parèntesis agrupen termes. Per claredat, s'agrupen per l'esquerra:

$\$(\$ a b c d \backslash equiv (((a b) c) d). \$\$$

La λ amb el punt introduceix funcions. Per claredat, es poden agrupar λ s: $\$(\$ \lambda x . \lambda y . a \backslash equiv \lambda x . (\lambda y . a) \backslash equiv \lambda xy.a \$\$$

6 / 39

Operacions

Només hi ha dues operacions per la construcció de termes:

- L'abstracció capture la idea de definir una funció amb un paràmetre: $\lambda x . u$ on u és un terme.

Diem que λx és el *cap* i que u és el *cos*.

Intuïció: $f(x, y) = x^2 + 2y + x - 1$ és representat per $\lambda x . \lambda y . x^2 + 2y + x - 1$

- L'aplicació capture la idea d'aplicar una funció sobre un paràmetre: $f \ x$ on f i x són dos termes.

Intuïció: $f(x)$ és representat per $f \ x$

7 / 39

Currificació

Al λ -càlcul totes les funcions tenen un sol paràmetre.

Les funcions que normalment consideraríem que tenen més d'un paràmetre es representen com a funcions d'un sol paràmetre utilitzant la tècnica del *currying*:

- Una funció amb dos paràmetres, com ara la suma, $+ : \text{int} \times \text{int} \rightarrow \text{int}$, es pot considerar equivalent a una funció d'un sol paràmetre que retorna una funció d'un paràmetre, $+ : \text{int} \rightarrow (\text{int} \rightarrow \text{int})$.
- Això vol dir que $2 + 3$, amb notació prefixa $(+ 2 \sim 3)$, s'interpretaria com $(+2) \sim 3$, on $(+2)$ és la funció que aplicada a qualsevol paràmetre x , retorna $x+2$.

Curricular és transformar una funció que accepta n paràmetres i convertir-la en una funció que, donat un paràmetre (el primer) retorna una funció que accepta $n-1$ paràmetres (i són semànticament equivalents).

8 / 39

Contingut

- Introducció
- Estructura bàsica
 - Components
 - Avaluació
 - Macros
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

9 / 39

Computació

La β -reducció (*cut-and-paste*) és la regla essencial de computació del λ -càlcul:

$\$(\lambda x . u) \setminus v \longrightarrow_{\beta} u[x:=v] \$$

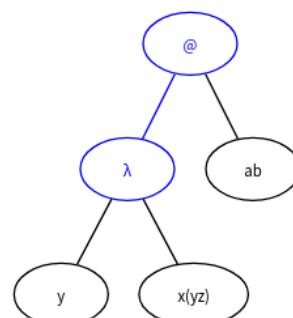
on $\$u[x:=v]\$$ vol dir reescriure $\$u\$$ substituint les seves $\$x\$$ per $\$v\$$.

Exemple:

Patró de la β -reducció.

expressió	acció efectuada
$\$(\lambda y. x(yz))(ab)\$$	β -reducció de $\$y\$$
$\$x((ab)z)\$$	$\$ \equiv \$$
$\$x(abz)\$$	

No cal que aparegui el patró a l'arrel
de l'arbre.



Forma normal

Si una expressió no pot β -reduir-se, aleshores es diu que està en forma normal.

Si $t \rightarrow \dots \rightarrow t' i t'$ està en forma normal, aleshores es diu que t' és la forma normal de t , i es considera que t' es el resultat de l'avaluació de t .

Una λ -expressió té, com a màxim, una forma normal.

11 / 39

Variables lliures i lligades

Dins d'un terme, una variable és lligada si apareix al cap d'una funció que la conté. Altrament és lliure.

Les variables poden ser lliures i lligades alhora en un mateix terme.

Per exemple: $\$(\lambda x. xy)(\lambda y. y)\$$

- y és lliure a la primera subexpressió.
- y és lligada a la segona subexpressió.

12 / 39

El problema de la captura de noms I

Quan s'aplica la β -reducció s'ha de tenir cura amb els noms de les variables i, si cal, reanomenar-les.

El problema es pot veure en el següent exemple: Sigui $\text{TWICE} = \lambda f. \lambda x. f(f x)$

Calculem $(\text{TWICE}) \sim (\text{TWICE})$:

expressió	acció efectuada
$\text{TWICE} \sim \text{TWICE}$	definició de TWICE
$(\lambda f. \lambda x. f(f x)) \sim (\lambda f. \lambda x. f(f x))$	β -reducció de f
$(\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x))) \sim (\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x)))$	definició de TWICE
$(\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x))) \sim (\lambda x. \text{TWICE}(\lambda f. \lambda y. f(f y)))$	

13 / 39

El problema de la captura de noms II

Aplicant la β -reducció directament tindríem:

expressió	acció efectuada
$(\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x))) \sim (\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x)))$	β -reducció de f
$(\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x))) \sim (\lambda x. \text{TWICE}(\lambda x. x(x)))$	ERROR

El que hauríem de fer és reanomenar la variable lligada x mes interna:

expressió	acció efectuada
$(\lambda x. \text{TWICE}(\lambda f. \lambda x. f(f x))) \sim (\lambda x. \text{TWICE}(\lambda f. \lambda y. f(f y)))$	canvi de nom $x \rightarrow y$
$(\lambda x. \text{TWICE}(\lambda f. \lambda y. f(f y))) \sim (\lambda x. \text{TWICE}(\lambda y. x(x y)))$	β -reducció de f
$(\lambda x. \text{TWICE}(\lambda y. x(x y))) \sim (\lambda y. x(x y))$	OK

14 / 39

α -Conversió

A més de la β -reducció, al λ -càlcul tenim la regla de l' α -conversió per reanomenar les variables. Per exemple:

$$\$ \$ \lambda x . \lambda y . xy \longrightarrow_a \lambda z . \lambda y . zy \longrightarrow_a \lambda z . \lambda t . zt \$ \$$$

Aleshores l'exemple del TWICE el podríem escriure:

expressió	acció efectuada
$\text{TWICE} \sim \text{TWICE}$	definició de TWICE
$(\lambda f. \lambda x. f(f x)) \text{TWICE}$	β -reducció de f
$(\lambda x. \text{TWICE})(\text{TWICE} \sim x)$	definició de TWICE
$(\lambda x. \text{TWICE})(\lambda f. \lambda x. f(f x)) \sim x$	α -conversió $[x/y]$
$(\lambda x. \text{TWICE})((\lambda f. \lambda y. f(f y)) \sim x) \backslash \backslash$	β -reducció de f
$(\lambda x. \text{TWICE})((\lambda y. x(x y)))$	

15 / 39

Ordres de reducció

Donada una λ -expressió, pot haver més d'un lloc on es pot aplicar β -reducció, per exemple:

$$\$ \$ (1) \sim (\lambda x . x((\lambda z . zz)x)) t \longrightarrow t((\lambda z . zz)t) \longrightarrow t(tt) \$ \$$$

però també:

$$\$ \$ (2) \sim (\lambda x . x((\lambda z . zz)x)) t \not\longrightarrow (\lambda x . x(xx)) t \longrightarrow t(tt) \$ \$$$

Hi ha dues formes estàndard d'avaluar una λ -expressió:

- Evaluació en ordre normal: s'aplica l'estratègia left-most outer-most:
Reduir la λ sintàcticament més a l'esquerra (1).
- Evaluació en ordre aplicatiu: s'aplica l'estratègia left-most inner-most:
Reduir la λ més a l'esquerra de les que són més endins (2).

16 / 39

Ordres de reducció

En principi, podríem pensar que no importa l'ordre d'avaluació que utilitzem, perquè la β -reducció és confluent:

Si $t \rightarrow \dots \rightarrow t_1$ i $t \rightarrow \dots \rightarrow t_2$ llavors
 $t_1 \rightarrow \dots \rightarrow t_3$ i $t_2 \rightarrow \dots \rightarrow t_3$

Tanmateix, si una expressió té una forma normal, aleshores la reducció en ordre normal la trobarà, però no necessàriament la reducció en ordre aplicatiu.

Per exemple, en ordre normal tenim:

$\$(\lambda x.a)((\lambda y.yy)(\lambda z.zz)) \rightarrow a\$$

però en ordre aplicatiu:

$\$(\lambda x.a)((\lambda y.yy)(\lambda z.zz)) \rightarrow (\lambda x.a)((\lambda z.zz)(\lambda z.zz)) \rightarrow \dots \$$

17 / 39

Contingut

- Introducció
- Estructura bàsica
 - Components
 - Avaluació
 - Macros
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

18 / 39

Macros

En el λ -càlcül, les funcions no reben noms.

Per facilitar-ne la escriptura, utilitzarem macros que representen funcions i les expandirem quan calgui, com vam fer a les transparències anteriors amb $\text{\textbackslash text\{TWICE\}}$.

Les macros també es diuen combinadors.

⇒ És un recurs "meta" que no forma part del llenguatge (preprocessador).

Exemple: $\$\\text\{ID\} \equiv \lambda x.x\$$

Llavors:

expressió	acció efectuada
$\$\\text\{ID\}\\ \\text\{ID\}$	definició $\$\\text\{ID\}$
$\$(\lambda x.x)\\ \\text\{ID\}\\ \\backslash \\backslash \$$	β -reducció de $\$x\$$
$\$\\text\{ID\}$	

19 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
 - Booleans
 - Naturals
 - Enters
- Recursivitat
- Universalitat
- Calculadores

20 / 39

Codificació de Church

Com es representen els tipus dades i els seus operadors en λ -càlcül.

Naturals en λ -càlcül: Una codificació estranya?

Dec	Bin	Romà	Xinès	Devanagari	λ -càlcül
0	0		零	०	$\lambda sz.z\$$
1	1	I	一	१	$\lambda sz.sz\$$
2	10	II	二	२	$\lambda sz.s(sz)\$$
3	11	III	三	३	$\lambda sz.s(s(sz))\$$
4	100	IV	四	४	$\lambda sz.s(s(s(sz)))\$$
\vdots				\vdots	

El natural n és l'aplicació n cops la funció s a z .

L'important no és com es representen els naturals, sinó establir una biecció entre la seva representació i \mathbb{N} .

Tampoc estem considerant-ne l'eficiència.

21 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
 - Booleans
 - Naturals
 - Enters
- Recursivitat
- Universalitat
- Calculadores

22 / 39

Booleans I

Church encoding*:

- $\$T \equiv \lambda t. \lambda f. t\$$ el primer
- $\$F \equiv \lambda t. \lambda f. f\$$ el segon

Com fem el $\$not\$$?

- $\$not \equiv \lambda g. gFT\$$ "flip"
- $\$not \equiv T \rightarrow (\lambda g. gFT)T \rightarrow TFT \rightarrow \dots \rightarrow F\$$ el primer
- $\$not \equiv F \rightarrow (\lambda g. gFT)F \rightarrow FFT \rightarrow \dots \rightarrow T\$$ el segon

Exercici: completar les β -reduccions.

* Church encoding - Wikipedia

23 / 39

Booleans II

Com fem el condicional?

- $\$if \equiv \lambda c. \lambda x. \lambda y. cxy\$$ el 1er o el 2on?
- Exercicis: codificar i avaluar:
 - if F then poma else pera
 - if T then poma else pera

Com fem l' $\$and\$$?

- $and \ x \ y = if \ x \ then \ y \ else \ F$
- $\$and \equiv \lambda x. \lambda y. xyF\$$
- Exercici: demostreu l'anterior.

I l' $\$or\$$?

- ...

24 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
 - Booleans
 - Naturals
 - Enters
- Recursivitat
- Universalitat
- Calculadores

25 / 39

Funcions aritmètiques bàsiques

- $\$0 \equiv \lambda f. \lambda x. x \equiv F$
- $\$n \equiv \lambda f. \lambda x. f^n x$

Com fem el $\$succ$?

- $\$succ = \lambda n. \lambda f. \lambda x. f(nfx)$ 1a f per afegir, fx per consumir
- Exercici: avalueu
 - $\$succ\ 1$

Com fem la $\$sum$?

- $\$sum \equiv \lambda m. \lambda n. n \ succ\ m$ $\$succ$ per cada f de la n
- Exercici: avalueu aplica n vegades $\$succ$ a m
 - $\$sum\ 2\ 1$

26 / 39

Més funcions aritmètiques

Altres:

- $\text{mul} \equiv \lambda m. \lambda n. \lambda f. n(mf)$

- $\text{power} \equiv \lambda m. \lambda n. nm$

- Exercici:

penseu el perquè, interpreteu-les

Avançats:

- $\text{minus} \equiv \lambda m. \lambda n. n \setminus \text{pred} \setminus m$

- $\text{pred} \equiv \lambda n. \lambda f. \lambda x. n(\lambda g. \lambda h. h(gf))(\lambda u. x)(\lambda u. u)$

27 / 39

Predicats

isZero?

- $\text{isZero} \equiv \lambda n. n(\lambda x. F)T$

0 consumeix F i és queda la T

- Exercici: avalueu

n es queda F i $\lambda x. F$ descarta la resta

- $\text{isZero} \setminus 0$

- $\text{isZero} \setminus 2$

Relacionals:

- $\text{leq} \equiv \lambda m. \lambda n. \text{IsZero} \setminus (\text{minus} \setminus m \setminus n)$

- $\text{eq} \equiv \lambda m. \lambda n. \text{and} \setminus (\text{leq} \setminus m \setminus n) (\text{leq} \setminus m \setminus n)$

28 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
 - Booleans
 - Naturals
 - Enters
- Recursivitat
- Universalitat
- Calculadores

29 / 39

Tuples

Parells:

- \$pair\equiv \lambda x.\lambda y.\lambda p.pxy\$
- Exemple: \$pair\ 2\ 3\equiv \lambda p.p\ 2\ 3\$

Accés:

- \$first\equiv \lambda p.p(\lambda x.\lambda y.x)\$
- \$second\equiv \lambda p.p(\lambda x.\lambda y.y)\$

Exercici: avalueu

- \$first\ (pair\ 2\ 3)\$
- \$second\ (pair\ 2\ 3)\$

30 / 39

Enters

Els codifiquem amb una resta en un parell:

$2 = \text{pair } 2 \ 0$
 $-3 = \text{pair } 0 \ 3$

Funcions:

- $\$convert \equiv \lambda x. \text{pair} \ x \ 0 \$$ natural a enter
- $\$neg \equiv \lambda x. \text{pair} \ (\text{second} \ x) \ (\text{first} \ x) \$$

S'utilitza una funció $\$oneZero\$$ per generar parells amb almenys un zero (amb recursivitat, $\$Y\$$).

Totes les funcions aritmètiques es generen tenint el compte els parells.

De la mateixa forma els racionals són parells d'enters.

Les llistes també es codifiquen a partir de parells (com en Lisp).

31 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

32 / 39

Recursivitat

Combinador Y (paradoxal o de punt fix):

$$\$\$Y\equiv \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))\$\$$$

Compleix la propietat:

$$\$\$YR\equiv R(YR)\$\$$$

Demostració:

expressió	acció efectuada
$\$\\text{Y} \\backslash \\text{R}$	definició de \$Y\$
$\$(\\lambda y . (\\lambda x.y(xx))(\\lambda x.y(xx)))\\text{R} \\backslash \\backslash \\$$	β -reducció de \$y\$
$\$(\\lambda x.\\text{R}(xx))(\\lambda x.\\text{R}(xx))$	β -reducció de \$x\$
$\\text{R}((\\lambda x.\\text{R}(xx))(\\lambda x.\\text{R}(xx)))$	per aquest resultat i l'anterior
$R(YR)$	

33 / 39

Factorial I

El combinador Y ens permet definir la funció factorial. Sigui:

$$\$\$H\equiv \lambda f.\lambda n.\\text{IF} \{n=0\} \sim 1 \sim (n \times (f \sim (n-1)))\$\$$$

podem veure com \$H\$ funciona com el factorial:

expressió	acció efectuada
\$H 1\$	combinador \$Y\$
\$H(H) 1\$	definició de \$H\$
$\$(\\lambda f.\\lambda n.\\text{IF} \{n=0\} 1 (n \times (f (n-1)))) (H) \\backslash \\backslash \\$$	β -reducció de \$f\$
$\$(\\lambda n.\\text{IF} \{n=0\} 1 (n \times (H (n-1)))) \\backslash 1 \\$$	β -reducció de \$n\$
$\\text{IF} \{1=0\} 1 (1 \\times (H (1-1)))$	$\text{IF}=\\text{fals}$
$1 \times (H (1-1))$	trivial
$H 0$	combinador \$Y\$
...	

34 / 39

Factorial II

expressió	acció efectuada
\$Y H 0\$	combinador \$Y\$
\$H (Y H) 0\$	definició de \$H\$
$\$ \lambda f. \lambda n. \text{text}\{IF\} (n=0) 1 (n \times (f (n-1))) (Y H) \ 0 \\ \$$	β -reducció de \$f\$
$\$ \lambda n. \text{text}\{IF\} (n=0) 1 (n \times (Y H (n-1))) \ 0 \$$	β -reducció de \$n\$
$\$ \text{text}\{IF\} (0=0) 1 (0 \times (Y H (0-1))) \$$	\$IF=cert\$
\$1\$	

35 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
- Recursivitat
- Universalitat
- Calculadores

36 / 39

Universalitat del λ -càcul

A partir d'aquí, ja només queda anar continuant fent definicions i anar-les combinant.

Eventualment, es pot arribar a veure que qualsevol algorisme és implementable en λ -càcul perquè pot simular a una màquina de Turing.

Teorema [Kleene i Rosser, 1936]: Totes les funcions recursives poden ser representades en λ -càcul (\Rightarrow Turing complet).

A diferència de les màquines de Turing que són un model matemàtic d'una màquina *hardware* imperativa, el λ -càcul només utilitza reescriptura i és un model matemàtic més *software* i funcional.

λ -càcul amb tipus: existeixen extensions amb tipus; que són les que solen utilitzar els llenguatges funcionals com model.

37 / 39

Contingut

- Introducció
- Estructura bàsica
- Codificacions de Church
- Recursivitat
- Universalitat
- [Calculadores](#)

38 / 39

Calculadores

Existeixen moltes calculadores de λ -càlcul *online*:

- https://www.cl.cam.ac.uk/~rmk35/lambda_calculus/lambda_calculus.html
- <https://jacksongl.github.io/files/demo/lambda/index.htm>
- <http://www-cs-students.stanford.edu/~blynn/lambda/> (amb notació Haskell)

Llenguatges de Programació

Programació funcional en Haskell



Jordi Petit i Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 2

Contingut

- Sessió 1: **recursivitat**

Aperitiu. Eines. Tipus bàsics. Funcions. Tuples. Llistes. Funcions habituals en llistes. Exercicis.

- Sessió 2: **funcions d'ordre superior**

Funcions d'ordre superior. Funcions d'ordre superior habituals. Aplicacions. Exercicis.

- Interludi: **aplicació vs composició**

Aplicació vs composició. Notació *point-free*. Exemple amb 2 paràmetres.

- Sessió 3: **llistes infinites**

Llistes per comprensió. Avaluació mandrosa. Llistes infinites. Exercicis.

- Sessió 4: **tipus algebraics i classes**

Tipus. Tipus algebraics. Tipus genèrics predefinitos. Classes. Exercicis.

- Sessió 5: **mònades**

Functors. Aplicatius. Mònades. Entrada/sortida. Exercicis.

- Sessió 6: **més mònades**

Mònade llista. Mònade estat. Combinació de mònades.

2 / 2

Llenguatges de Programació

Sessió 1: Recursivitat



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



1 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

2 / 62

Haskell

Haskell és llenguatge de programació funcional pura.

No hi ha:

- assignacions,
- bucles,
- efectes laterals,
- gestió explícita de la memòria.

Hi ha:

- evaluació *lazy*,
- funcions com a objectes de primer ordre,
- sistema de tipus estàtic,
- inferència de tipus automàtica.

Haskell és elegant, concís i fa pensar d'una forma diferent!

3 / 62

Expressions

$\lambda> 3 + 2 * 2$
👉 7

$\lambda> (3 + 2) * 2$
👉 10

$\lambda> \text{even } 42$
👉 True

$\lambda> \text{even}(42)$ -- 🚫 parèntesis absurds
👉 True

$\lambda> \text{even "Arnau"}$ -- ✗ error de tipus

$\lambda> \text{div } 14 \ 4$
👉 3

4 / 62

Tipus

```
λ> :type 'R'  
👉 'R' :: Char  
  
λ> :type "Marta"  
👉 "Marta" :: [Char]  
  
λ> :type not  
👉 not :: Bool -> Bool  
  
λ> :type length  
👉 length :: [a] -> Int
```

5 / 62

Factorial

```
factorial :: Integer -> Integer  
factorial 0 = 1  
factorial n = n * factorial (n - 1)  
  
λ> factorial 5  
👉 120  
  
λ> map factorial [0..5]  
👉 [1, 1, 2, 6, 24, 120]
```

6 / 62

Quicksort

```
quicksort []      = []
quicksort (p:xs) = (quicksort menors) ++ [p] ++ (quicksort majors)
  where
    menors = [x | x <- xs, x < p]
    majors = [x | x <- xs, x >= p]
```

```
λ> :type quicksort
👉 quicksort :: Ord t => [t] -> [t]
```

```
λ> quicksort [5, 3, 6, 3, 1]
👉 [1, 3, 3, 5, 6]
```

```
λ> quicksort ["joan", "sara", "pep", "jana"]
👉 ["jana", "joan", "pep", "sara"]
```

7 / 62

Arbres binaris

```
data Arbin t = Buit
             | Node t (Arbin t) (Arbin t)

alcada :: Arbin t -> Integer
alcada Buit = 0
alcada (Node x fe fd) = 1 + max (alcada fe) (alcada fd)

preordre :: Arbin t -> [t]
preordre Buit = []
preordre (Node x fe fd) = [x] ++ preordre fe ++ preordre fd
```

8 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

9 / 62

Eines necessàries

Glasgow Haskell Compiler (GHC):

- compilador (`ghc`)
- intèrpret (`ghci`)

Editor de codi

Terminal

Jutge

Instal·lació del GHC

Linux i Mac:

Utilitzeu el `ghcup` (referència):

```
curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
```

Windows:

Seguiu aquest [vídeo](#).

Font: [GHCup: How to install](#)

11 / 62

Jutge

Apunteu-vos al curs "Problems in Haskell" de [Jutge.org](#).

The screenshot shows a web browser window with the URL <https://jutge.org/problems>. The page title is "Problems". Below it, there are tabs for "By course", "Accepted", "Rejected", "All", and "Search". A search bar is present. The main content area is titled "Problems in Haskell" and lists problems categorized by lab:

- Lab 1**:
 - P17607 Haskell - Functions with numbers
 - P50084 Haskell - Functions with lists
 - P50040 Haskell - Sorting
- Lab 2**:
 - P90032 Haskell - Usage of higher-order functions (1)
 - P51745 Haskell - Usage of higher-order functions (2)
 - P50688 Haskell - Usage of comprehension lists
- Lab 3**:
 - P90077 Haskell - Definition of higher-order functions (1)
 - P71775 Haskell - Definition of higher-order functions (2)
 - P95857 Haskell - Infinite lists (1)
- Lab 4**:
 - P97301 Haskell - FizzBuzz
 - P51072 Haskell - Binary tree
 - P40618 Haskell - Queue (1)
- Lab 5**:
 - P70640 Haskell - Expressions
 - P50086 Haskell - Cosa (2)
- Lab 6**:
 - P82074 Haskell - Hello / Bye
 - P57082 Haskell - Body mass index
- Past exams**:
 - P40646 Haskell - Pascal 2017-12-04
 - P51910 Haskell - Pascal 2018-04-11
 - P52038 Haskell - Pascal 2018-11-06
 - Pascal 2019-05-05
 - P53322 Haskell - Números molt compostos



12 / 62

Comandes de l'intèpret

Intèpret:

ghci.

Comandes més usuals:

Comanda	Exemple	Descripció
:load	:l arxiu	càrrega un script
:quit	:q	sortida de l'intèpret
:reload	:r	recarrega l'últim arxiu carregat
:type	:t 3	tipus de l'expressió
:info	:i []	informació associada al paràmetre (útil a partir del tema de classes)
:sprint		visualització dels <i>thunks</i> (útil per l'avaluació mandrosa)
:help		ajuda

13 / 62

Contingut

- Aperitiu
- Eines
- [Tipus bàsics](#)
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

14 / 62

Booleans

Tipus: Bool

Literals: False i True

Operacions:

<code>not</code>	:: Bool -> Bool	-- negació
<code>()</code>	:: Bool -> Bool -> Bool	-- disjunció
<code>(&&)</code>	:: Bool -> Bool -> Bool	-- conjunció

Exemples:

<code>not True</code>	👉 False
<code>not False</code>	👉 True
<code>True False</code>	👉 True
<code>True && False</code>	👉 False
<code>(False True) and True</code>	👉 True
<code>not (not True)</code>	👉 True
<code>not not True</code>	✗ -- vol dir: (not not) True

15 / 62

Enters

Tipus:

- Int: Enters de 64 bits en Ca2
- Integer: Enters arbitràriament llargs

Literals: 16, ! (-22), 587326354873452644428

Operacions: +, -, *, div, mod, rem, ^.

Operadors relacionals: <, >, <=, >=, ==, /= (! no !=)

Exemples:

<code>3 + 4 * 5</code>	👉 23
<code>(3 + 4) * 5</code>	👉 35
<code>(3 + 4) * 5</code>	👉 35
<code>2^10</code>	👉 1024
<code>3 + 1 /= 4</code>	👉 False
<code>div 11 2</code>	👉 5
<code>mod 11 2</code>	👉 1
<code>rem 11 2</code>	👉 1
<code>mod (-11) 2</code>	👉 1
<code>rem (-11) 2</code>	👉 -1

16 / 62

Reals

Tipus:

- `Float`: Reals de coma flotant de 32 bits
- `Double`: Reals de coma flotant de 64 bits

Literals: `3.14`, `1e-9`, `-3.0`

Operacions: `+`, `-`, `*`, `/`, `**`.

Operadors relacionals: `<`, `>`, `<=`, `>=`, `==`, `/=`

Conversió enter a real: `fromIntegral`

Conversió real a enter: `round`, `floor`, `ceiling`

Exemples:

```
10.0 / 3.0      ↗ 3.333333333333335
2.0 ** 3.0      ↗ 8.0
fromIntegral 4  ↗ 4.0
```

17 / 62

Caràcters

Tipus: `Char`

Literals: `'a'`, `'A'`, `'\n'`

Operadors relacionals: `<`, `>`, `<=`, `>=`, `==`, `/=`

Funcions de conversió: (cal un `import Data.Char`)

- `ord :: Char -> Int`
- `chr :: Int -> Char`

18 / 62

Precedència dels operadors

Precedència	Associatius per l'esquerra	No associatius	Associatius per la dreta
9	!!		.
8			^, ^^, **
7	* / div mod rem quot		
6	+ -		
5			: ++
4		== /= < <= > >=	
		elem notElem	
3			&&
2			
1	>> >>=		
0			\$ \$! seq

Font: [Haskell report](#)

19 / 62

Funcions predefinides habituals

és parell/senar:

```
even :: Integral a => a -> Bool  
odd :: Integral a => a -> Bool
```

mínim i màxim de dos valors:

```
min :: Ord a => a -> a -> a  
max :: Ord a => a -> a -> a
```

màxim comú divisor, mínim comú múltiple:

```
gcd :: Integral a => a -> a -> a  
lcm :: Integral a => a -> a -> a
```

matemàtiques:

```
abs :: Num a => a -> a  
sqrt :: Floating a => a -> a  
log :: Floating a => a -> a  
exp :: Floating a => a -> a  
cos :: Floating a => a -> a
```

20 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

21 / 62

Transparència referencial

- Les funcions en Haskell són *pures*: només retornen resultats calculats en relació als seus paràmetres.
- Les funcions no tenen efectes laterals (*side effects*).
 - no modifiquen els paràmetres
 - no modifiquen la memòria
 - no modifiquen l'entrada/sortida
- Una funció sempre retorna el mateix resultat aplicada sobre els mateixos paràmetres.

22 / 62

Definició de funcions

Els identificadors de funcions comencen amb minúscula.

Per introduir una funció:

1. Primer es dóna la seva declaració de tipus (capçalera).
2. Després es dóna la seva definició, utilitzant paràmetres formals.

Exemples:

```
doble :: Int -> Int           -- calcula el doble d'un valor
doble x = 2 * x

perimetre :: Int -> Int -> Int      -- calcula l'àrea d'un rectangle
perimetre amplada alçada = doble (amplada + alçada)

xOr :: Bool -> Bool -> Bool        -- o exclusiva
xOr a b = (a || b) && not (a && b)

factorial :: Integer -> Integer     -- calcula el factorial d'un natural
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

23 / 62

Definicions amb patrons

Les funcions es poden definir amb patrons:

```
factorial :: Integer -> Integer
-- calcula el factorial d'un natural

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

L'avaluació dels patrons és de dalt a baix i retorna el resultat de la primera branca que casa.

Els patrons es consideren més elegants que el `if-then-else` i tenen moltes més aplicacions.

`_` representa una variable anònima: (no hi ha relació entre diferents `_`)

```
nand :: Bool -> Bool -> Bool      -- conjunció negada
nand True True = False
nand _ _ = True
```

24 / 62

Definicions amb guardes

Les funcions es poden definir amb guardes:

```
valAbs :: Int -> Int
-- retorna el valor absolut d'un enter

valAbs n
| n >= 0    = n
| otherwise = -n
```

L'avaluació de les guardes és de dalt a baix i retorna el resultat de la primera branca certa. (Error si cap és certa)

Les definicions per patrons també poden tenir guardes.

El `otherwise` és el mateix que `True`, però més llegible.

⚠ La igualtat va després de cada guarda!

25 / 62

Definicions locals

Per definir noms locals en una expressió s'utilitza el `let-in`:

```
fastExp :: Integer -> Integer -> Integer      -- exponenciació ràpida
fastExp _ 0 = 1
fastExp x n =
  let y = fastExp x n_halved
      n_halved = div n 2
  in
    if even n
    then y * y
    else y * y * x
```

El `where` permet definir noms en més d'una expressió:

```
fastExp :: Integer -> Integer -> Integer      -- exponenciació ràpida
fastExp _ 0 = 1
fastExp x n
  | even n    = y * y
  | otherwise = y * y * x
  where
    y = fastExp x n_halved
    n_halved = div n 2
```

La identació del `where` defineix el seu àmbit.

26 / 62

Currificació

Totes les funcions tenen un únic paràmetre.

Les funcions de més d'un paràmetre retornen, en realitat, una nova funció.

No cal passar tots els paràmetres (aplicació parcial).

Exemple:

`prod 3 5` és, en realitat, `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

```
prod :: Int -> (Int -> Int)
```

```
(prod 3) :: (Int -> Int)
```

```
(prod 3) 5 :: Int
```

27 / 62

Inferència de tipus

Si no es dóna la capçalera d'una funció, Haskell infereix el seu tipus.

Amb aquestes definicions,

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

Haskell infereix que `factorial :: Num t => t -> t`.

Es pot preguntar el tipus d'una expressió amb :type a l'intèrpret:

```
λ> :type factorial
👉 factorial :: Num t => t -> t
```

💬 Al principi, no useu la inferència de tipus (generalitza massa i perdeu disciplina).

💬 Pels problemes del Jutge, copieu les capçaleres donades als exercicis.

28 / 62

Notació prefixa/infixa

$\frac{2+3}{(+)} \quad 2 \quad 3$

 5
 5

Els operadors són infixes \Rightarrow posar-los entre parèntesis per fer-los prefixes

$\frac{\text{div } 9 \ 4}{9 \ ` \text{div} ` \ 4} \quad 9 \quad 4$

 2
 2

Les funcions són prefixes \Rightarrow posar-les entre *backticks* per fer-les infixes

29 / 62

Sumari

- Les funcions en Haskell tenen un sol paràmetre (currificació).
 - $a \rightarrow b \rightarrow c$ vol dir $a \rightarrow (b \rightarrow c)$.
 - $f \ x \ y$ vol dir $(f \ x) \ y$.
- Per escriure una funció cal donar
 - la seva capçalera i
 - la seva definició.
- La inferència de tipus evita descriure les capçaleres de les funcions. Eviteu-la al principi.
- Les definicions poden ser úniques o amb patrons i cada definició pot tenir guardes.
- Els patrons i les guardes es trien de dalt a baix.
- Es poden crear definicions locals amb el `let` i el `where` i es poden usar patrons localment amb el `case`.

30 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- **Tuples**
- Llistes
- Funcions habituals en llistes
- Exercicis

31 / 62

Tuples

Una tupla és un tipus estructurat que permet desar diferents valors de tipus t_1, t_2, \dots, t_n en un únic valor de tipus (t_1, t_2, \dots, t_n) .

- El nombre de camps és fix.
- Els camps són de tipus heterogenis.

```
(3, 'z', False) :: (Int, Char, Bool)
(6, 9)      :: (Int, Int)
(True, (6, 9)) :: (Bool, (Int, Int))
```

```
caracterMesFrequent :: String -> (Char, Int)
caracterMesFrequent "PATATA" ↗ ('A', 3)
```

```
descomposicioHoraria :: Int -> (Int, Int, Int)    -- hores, minuts, segons
descomposicioHoraria segons = (h, m, s)
  where
    h = div segons 3600
    m = div (mod segons 3600) 60
    s = mod segons 60
```

32 / 62

Accés a tuples

Per a tuples de dos elements, es pot accedir amb `fst` i `snd`:

```
fst :: (a, b) -> a  
snd :: (a, b) -> b
```

```
fst (3, "rave")      ↗ 3  
snd (3, "rave")      ↗ "rave"
```

Per a tuples generals, no hi ha definides funcions d'accés

⇒ Es poden crear fàcilment usant patrons:

```
primer (x, y, z) = x  
segon (x, y, z) = y  
tercer (x, y, z) = z
```

```
primer (x, _, _) = x  
segon (_, y, _) = y  
tercer (_, _, z) = z
```

33 / 62

Descomposició de tuples en patrons

Lleig:

```
distancia :: (Float, Float) -> (Float, Float) -> Float  
-- calcula la distància entre dos punts 2D, cadascun donat amb una tupla  
  
distancia p1 p2 = sqrt ((fst p1 - fst p2)^2 + (snd p1 - snd p2)^2)
```

Millor: Descompondre per patrons als propis paràmetres:

```
distancia (x1, y1) (x2, y2) = sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

També: Descompondre per patrons usant noms locals:

```
distancia p1 p2 = sqrt (sqr dx + sqr dy)  
where  
  (x1, y1) = p1  
  (x2, y2) = p2  
  dx = x1 - x2  
  dy = y1 - y2  
  sqr x = x * x
```

34 / 62

Tupla buida (*unit*)

Existeix el tipus de tupla sense cap dada, que només té un possible valor: la dada buida.

Concepte semblant al `void` del C.

- Tipus: ()
- Valor: ()

En algun moment en farem ús.

35 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

36 / 62

Llistes

Una llista és un tipus estructurat que conté una seqüència d'elements, tots del mateix tipus.

[t] denota el tipus de les llistes d'elements de tipus t.

```
[]                                -- llista buida  
[3, 9, 27]                         :: [Int]  
[(1, "un"), (2, "dos"), (3, "tres")] :: [(Int, String)]  
[[7], [3, 9, 27], [1, 5], []]        :: [[Int]]  
[1 .. 10]                            -- el mateix que [1,2,3,4,5,6,7,8,9,10]  
[1, 3 .. 10]                         -- el mateix que [1,3,5,7,9]
```

37 / 62

Constructors de llistes

Les llistes tenen dos constructors: [] i :

- La llista buida:

```
[] :: [a]
```

- Afegir per davant:

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

38 / 62

Constructors de llistes

La notació

[16, 12, 21]

és una drecera per

16 : 12 : 21 : []

que vol dir

16 : (12 : (21 : []))

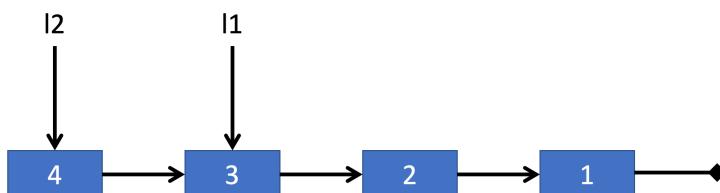
39 / 62

Implementació i eficiència

Les llistes de Haskell són llistes simplement encadenades.

Els constructors [] i : funcionen en temps constant (*DS sharing*).

l1 = 3 : 2 : 1 : []
l2 = 4 : l1



L'operador ++ retorna la concatenació de dues llistes (temps proporcional a la llargada de la primera llista).

40 / 62

Llistes i patrons

La discriminació per patrons permet descompondre les llistes:

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

Diem que $\$e_1\$$ matches $\$e_2\$$ si existeix una substitució per les variables de $\$e_1\$$ que la fan igual que $\$e_2\$$.

Exemples:

- $x:xs$ matches $[2, 5, 8]$ perquè $[2, 5, 8]$ és $2 : (5 : 8 : [])$ substituint x amb 2 i xs amb $(5 : 8 : [])$ que és $[5, 8]$.
- $x:xs$ does not match $[]$ perquè $[]$ i $:xs$ són constructors diferents.
- $x1:x2:xs$ matches $[2, 5, 8]$ substituint $x1$ amb 2 , $x2$ amb 5 i xs amb $[8]$.
- $x1:x2:xs$ matches $[2, 5]$ substituint $x1$ amb 2 , $x2$ amb 5 i xs amb $[]$.

Nota: El mecanisme de *matching* no és el mateix que el d'*unificació* (Prolog).

41 / 62

Llistes i patrons

La descomposició per patrons també es pot usar als `case`, `where` i `let`.

```
suma llista =
  case llista of
    []          -> 0
    x:xs       -> x + suma xs

divImod n m
  | n < m      = (0, n)
  | otherwise   = (q + 1, r)
  where (q, r) = divImod (n - m) m

primerIsegon llista =
  let primer:segon:resta = llista
  in (primer, segon)
```

42 / 62

Textos

Els textos (*strings*) en Haskell són llistes de caràcters.

El tipus `String` és una sinònim de `[Char]`.

Les cometes dobles són sucre sintàctic per definir textos.

```
nom1 :: [Char]  
nom1 = 'p':'e':'p':[]
```

```
nom2 :: String  
nom2 = "pepa"
```

```
λ> nom1 == nom2
```

👉 False

```
λ> nom1 < nom2
```

👉 True

43 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- [Funcions habituals en llistes](#)
- Exercicis

44 / 62

head, last

- Signatura:

```
head :: [a] -> a  
last :: [a] -> a
```

- Descripció:

- head xs és el primer element de la llista xs.
- last xs és el darrer element de la llista xs.

Error si xs és buida.

- Exemples:

```
λ> head [1..4]  
👉 1  
λ> last [1..4]  
👉 4
```

45 / 62

tail, init

- Signatura:

```
tail :: [a] -> [a]  
init :: [a] -> [a]
```

- Descripció:

- tail xs és la llista xs sense el seu primer element.
- init xs és la llista xs sense el seu darrer element.

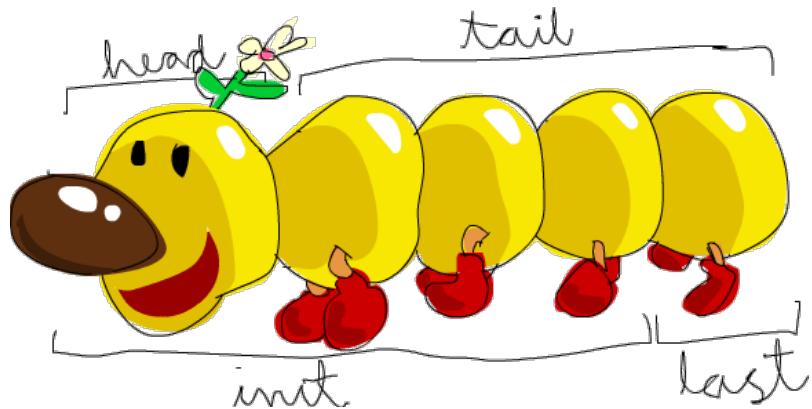
Error si xs és buida.

- Exemples:

```
λ> tail [1..4]  
👉 [2, 3, 4]  
λ> init [1..4]  
👉 [1, 2, 3]
```

46 / 62

head, last, init, tail



Dibuix: Learn You a Haskell, M. Lipovača

47 / 62

reverse

- Signatura:

`reverse :: [a] -> [a]`

- Descripció:

`reverse xs` és la llista `xs` del revés.

- Exemples:

```
λ> reverse [1..4]
👉 [4, 3, 2, 1]
```

48 / 62

length

- Signatura:

```
length :: [a] -> Int
```

- Descripció:

length xs és el nombre d'elements a la llista xs.

- Exemples:

```
λ> length []
👉 0
λ> length [1..5]
👉 5
λ> length "Marta"
👉 5
```

49 / 62

null

- Signatura:

```
null :: [a] -> Bool
```

- Descripció:

null xs indica si la llista xs és buida.

- Exemples:

```
λ> null []
👉 True
λ> null [1..5]
👉 False
```

50 / 62

elem

- Signatura:

```
elem :: Eq a => a -> [a] -> Bool
```

- Descripció:

elem x xs indica si x és a la llista xs.

- Exemples:

```
λ> elem 3 [1..10]
👉 True
λ> 3 `elem` [1..10]
👉 True
λ> 'k' `elem` "Jordi"
👉 False
```

51 / 62

Indexació: (! !)

- Signatura:

```
(!!) :: [a] -> Int -> a
```

- Descripció:

xs !! i és l'i-èsim element de la llista xs (començant per zero).

- Exemples:

```
λ> [1..10] !! 3
👉 4
λ> [1..10] !! 11
✗ Exception: index too large
```

52 / 62

Concatenació de dues llistes: (++)

- Signatura:

```
(++) :: [a] -> [a] -> [a]
```

- Descripció:

`xs ++ ys` és la llista resultant de posar `ys` darrera de `xs`.

- Exemples:

```
λ> "PEP" ++ "ET"  
👉 "PEPET"  
λ> [1..5] ++ [1..3]  
👉 [1,2,3,4,5,1,2,3]
```

53 / 62

maximum, minimum

- Signatura:

```
maximum :: Ord a => [a] -> a  
minimum :: Ord a => [a] -> a
```

- Descripció:

- `maximum xs` és l'element més gran de la llista (no buida!) `xs`.
- `minimum xs` és l'element més petit de la llista (no buida!) `xs`.

- Exemples:

```
λ> maximum [1..10]  
👉 10  
λ> minimum [1..10]  
👉 1  
λ> minimum []  
✗ Exception: empty list
```

54 / 62

sum, product

- Signatura:

```
sum      :: Num a => [a] -> a
product :: Num a => [a] -> a
```

- Descripció:

- `sum xs` és la suma de la llista `xs`.
 - `prod xs` és el producte de la llista `xs`.

- Exemples:

```
λ> sum [1..5]
👉 15
```

```
factorial n = product [1 .. n]
```

```
λ> factorial 5
👉 120
```

55 / 62

and, or

- Signatura:

```
and :: [Bool] -> Bool
or  :: [Bool] -> Bool
```

- Descripció:

- `and bs` és la conjunció de la llista de booleans `bs`.
 - `or bs` és la disjunció de la llista de booleans `bs`.

- Observació:

- Distingiu bé entre `and/or` i `(&&)/(||)`.

56 / 62

take, drop

- Signatura:

```
take :: Int -> [a] -> [a]  
drop :: Int -> [a] -> [a]
```

- Descripció:

- take n xs és el prefixe de llargada n de la llista xs .
- drop n xs és el sufixe de la llista xs quan se li treuen els n primers elements.

- Exemples:

```
λ> take 3 [1 .. 7]  
👉 [1, 2, 3]  
λ> drop 3 [1 .. 7]  
👉 [4, 5, 6, 7]
```

57 / 62

zip

- Signatura:

```
zip :: [a] -> [b] -> [(a, b)]
```

- Descripció:

zip xs ys és la llista que combina, en ordre, cada parell d'elements de xs i ys . Si en falten, es perden.

- Exemples:

```
λ> zip [1, 2, 3] ['a', 'b', 'c']  
👉 [(1, 'a'), (2, 'b'), (3, 'c')]  
λ> zip [1 .. 10] [1 .. 3]  
👉 [(1, 1), (2, 2), (3, 3)]
```

58 / 62

repeat

- Signatura:

```
repeat :: a -> [a]
```

- Descripció:

`repeat x` és la llista infinita on tots els elements són `x`.

- Exemples:

```
λ> repeat 3
👉 [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, ...]
λ> take 4 (repeat 3)
👉 [3, 3, 3, 3]
```

59 / 62

concat

- Signatura:

```
concat :: [[a]] -> [a]
```

- Descripció:

`concat xs` és la llista que concatena totes les llistes de `xs`.

- Exemples:

```
λ> concat [[1, 2, 3], [], [3], [1, 2]]
👉 [1, 2, 3, 3, 1, 2]
```

60 / 62

Contingut

- Aperitiu
- Eines
- Tipus bàsics
- Funcions
- Tuples
- Llistes
- Funcions habituals en llistes
- Exercicis

61 / 62

Exercicis

1. Instal·leu-vos les eines per treballar.
2. Proveu de cercar documentació de funcions a [Hoogλe](#).
3. Feu aquests problemes de Jutge.org:
 - [P77907 Functions with numbers](#)
 - [P25054 Functions with lists](#)
 - [P29040 Sorting](#)
 - Novetats:
 - Problemes amb puntuacions parcials [100](#). No cal que feu totes les funcions demandades.
 - Inspector de Haskell: comprova condicions de l'enunciat en el codi de la solució. Veredict NC *Non compliant*. [TFG d'en Jan Mas]
4. Implementeu les funcions habituals sobre llistes vistes anteriorment.
 - Useu notació tipus `myLength` enlloc de `length` per evitar xocs de noms.
 - Useu recursivitat quan calgui o useu altres funcions `my*` que ja hagueu definit.

62 / 62

Llenguatges de Programació

Sessió 2: funcions d'ordre superior



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 40

Contingut

- Funcions d'ordre superior
- Funcions d'ordre superior habituals
- Aplicacions
- Exercicis

2 / 40

Funcions d'ordre superior

Una funció d'ordre superior (FOS) és una funció que rep o retorna funcions.

Punt clau: les funcions són objectes de primera classe.

Exemple en C++:

```
bool compare(int x, int y) {
    return x > y;
}

int main() {
    vector<int> v = { ... };
    sort(v.begin(), v.end(), compare);           // sort és funció d'ordre superior
}
```

3 / 40

Funcions d'ordre superior

Exemples:

La funció predefinida `map` aplica una funció a cada element d'una llista.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
λ> map odd [1..5]
👉 [True, False, True, False, True]
```

La funció predefinida `(.)` retorna la composició de dues funcions:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

```
λ> (reverse . sort) [5, 3, 5, 2]
👉 [5, 5, 3, 2]
```

4 / 40

Funcions d'ordre superior

Exemple: La funció `apli2` aplica dos cops una funció a un element.

```
apli2 :: (a -> a) -> a -> a  
apli2 f x = f (f x)
```

```
λ> apli2 sqrt 16.0  
👉 2.0
```

De forma equivalent:

```
apli2 :: (a -> a) -> (a -> a)  
apli2 f = f . f
```

```
λ> apli2 sqrt 16.0  
👉 2.0
```

Petit exercici:

```
λ> per2 x = 2 * x  
λ> apli2 (apli2 per2) 2  
👉 ?
```

5 / 40

Funcions anònimes

Les funcions anònimes (funcions λ) són expressions que representen una funció sense nom.

```
\x -> x + 3          -- defineix funció anònima que, donada una x, retorna x + 3  
                      -- si proveu d'escriure-la, Haskell s'enfada perquè no ho sap  
(\x -> x + 3) 4      -- aplica la funció anònima sobre 4  
👉 7
```

Funció amb nom:

```
doble x = 2 * x          -- equival a doble = \x -> 2 * x  
λ> doble 3  
λ> map doble [1, 2, 3]    👈 6  
                           👈 [2, 4, 6]
```

Funció anònima:

```
λ> map (\x -> 2 * x) [1, 2, 3]    👈 [2, 4, 6]
```

Utilitat: quan són curtes i només s'utilitzen un cop.

També són útils per realitzar transformacions de programes.

6 / 40

Funcions anònimes

Múltiples paràmetres:

$\lambda x \ y \rightarrow x + y$

és equivalent a

$\lambda x \rightarrow (\lambda y \rightarrow x + y)$

que vol dir

$\lambda x \rightarrow (\lambda y \rightarrow x + y)$

7 / 40

Seccions

Les seccions permeten aplicar operadors infixos parcialment.

Per la dreta:

$(\otimes \ y) \equiv \lambda x \rightarrow x \otimes y$

Per l'esquerra:

$(y \otimes) \equiv \lambda x \rightarrow y \otimes x$

Exemples:

```
λ> doble = (* 2)
λ> doble 3
👉 6
```

```
λ> map (* 2) [1, 2, 3]    -- millor que map (\x -> x * 2) [1, 2, 3]
👉 [2, 4, 6]
```

```
λ> meitat = (/ 2)          -- ≠ (2 /)
λ> meitat 6
👉 3
```

```
λ> ésMajúscula = (`elem` ['A'..'Z'])
λ> ésMajúscula 'b'
👉 False
```

8 / 40

Contingut

- Funcions d'ordre superior
- Funcions d'ordre superior habituals
- Aplicacions
- Exercicis

9 / 40

Funcions d'ordre superior habituals

Algunes funcions d'ordre superior predefined s'utilitzen molt habitualment:

- (.)
- (\$)
- const
- id
- flip
- map
- filter
- zipWith
- all, any
- dropWhile, takeWhile
- iterate, until
- foldl, foldr
- scanl, scanr

10 / 40

composició (.)

- Signatura:

`(.) :: (b -> c) -> (a -> b) -> a -> c`

- Descripció:

`f . g` és la composició de les funcions `f` i `g`.

- Exemples:

```
λ> tresMesGrans = take 3 . reverse . sort
```

```
λ> :type tresMesGrans
tresMesGrans :: Ord a => [a] -> [a]
```

```
λ> tresMesGrans [3, 1, 2, 6, 7]
👉 [7, 6, 3]
```

11 / 40

aplicació (\$)

- Signatura:

`($) :: (a -> b) -> a -> b`

- Descripció:

`f $ x` és el mateix que `f x`. Sembla inútil, però degut a la baixa prioritat d'aquest operador, ens permet ometre molts parèntesis de tancar!

- Exemples:

```
λ> tail (tail (tail (tail "Jordi")))
👉 "i"
λ> tail $ tail $ tail $ tail "Jordi"
👉 "i"
```

12 / 40

const

- Signatura:

```
const :: a -> b -> a
```

- Descripció:

const x és una funció que sempre retorna x, independentment de què se li apliqui.

- Exemples:

```
λ> map (const 42) [1 .. 5]  
👉 [42, 42, 42, 42, 42]
```

13 / 40

id

- Signatura:

```
id :: a -> a
```

- Descripció:

id és la funció identitat. També sembla inútil, però va bé en algun moment.

- Exemples:

```
λ> map id [1 .. 5]  
👉 [1, 2, 3, 4, 5]
```

14 / 40

flip

- Signatura:

flip :: (a -> b -> c) -> (b -> a -> c)

- Descripció:

flip f retorna la funció f però amb els seus dos paràmetres invertits. Es defineix per

flip f x y = f y x

- Exemples:

```
λ> meitat = flip div 2
```

```
λ> meitat 10
```

👉 5

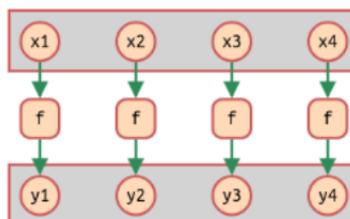
15 / 40

map

- Signatura:

map :: (a -> b) -> [a] -> [b]

- Descripció: **map f xs** és la llista que s'obté al aplicar la funció f a cada element de la llista xs, de forma que **map f [x₁, x₂, ..., x_n]** és [f x₁, f x₂, ..., f x_n].



[y₁, y₂, y₃, y₄] = **map f [x₁, x₂, x₃, x₄]**

- Exemples:

```
λ> map even [2, 4, 6, 7] 👉 [True, True, True, False]  
λ> map (*2) [2, 4, 6, 7] 👉 [4, 8, 12, 14]
```

16 / 40

filter

- Signatura:

```
filter :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

filter p xs és la subllista dels elements de xs que compleixen el predicat p .

(Un predicat és una funció que retorna un Booleà.)

- Exemples:

```
λ> filter even [2, 1, 4, 6, 7]  
👉 [2, 4, 6]
```

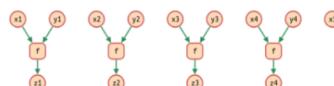
17 / 40

zipWith

- Signatura:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- Descripció: $\text{zipWith } op \ xs \ ys$ és la llista obtinguda operant cada element de xs amb cada element de ys via la funció op , d'esquerra a dreta, mentre n'hi hagi.



```
[z1, z2, z3, z4] = zipWith f [x1, x2, x3, x4, x5] [y1, y2, y3, y4]
```

- Exemples:

```
λ> zipWith (+) [1, 2, 3] [5, 1, 8, 9]  
👉 [6, 3, 11]
```

18 / 40

all

- Signatura:

```
all :: (a -> Bool) -> [a] -> Bool
```

- Descripció:

all p xs indica si tots els elements de xs compleixen el predicat p.

- Exemples:

```
λ> all even [2, 1, 4, 6, 7]  
👉 False  
λ> all even [2, 4, 6]  
👉 True
```

19 / 40

any

- Signatura:

```
any :: (a -> Bool) -> [a] -> Bool
```

- Descripció:

any p xs indica si algun dels elements de xs compleix el predicat p.

- Exemples:

```
λ> any even [2, 1, 4, 6, 7]  
👉 True  
λ> any odd [2, 4, 6]  
👉 False
```

20 / 40

dropWhile

- Signatura:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

`dropWhile p xs` és la subllista de `xs` que elimina els primers elements de `xs` que compleixen el predicat `p` (fins al final o al primer que no la compleix).

- Exemples:

```
λ> dropWhile even [2, 4, 6, 7, 8]  
👉 [7, 8]  
λ> dropWhile even [2, 4]  
👉 []
```

21 / 40

takeWhile

- Signatura:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

- Descripció:

`takeWhile p xs` és la subllista de `xs` que conté els primers elements de `xs` que compleixen el predicat `p` (fins al final o al primer que no la compleix).

- Exemples:

```
λ> takeWhile even [2, 4, 6, 7, 8]  
👉 [2, 4, 6]  
λ> takeWhile even [1, 3]  
👉 []
```

22 / 40

iterate

- Signatura:

```
iterate :: (a -> a) -> a -> [a]
```

- Descripció:

iterate f x retorna la llista infinita [x, f x, f (f x), f (f (f x)), ...].

```
ys = iterate f x
```

- Exemples:

```
λ> iterate (*2) 1  
👉 [1, 2, 4, 8, 16, ...]
```

23 / 40

until

- Signatura:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
```

- Descripció: until p f x retorna la llista [x, f x, f (f x), f (f (f x)), ...] fins que es satisfa el predicat p.

- Exemples:

```
λ> until (>100) (*3) 1  
👉 243
```

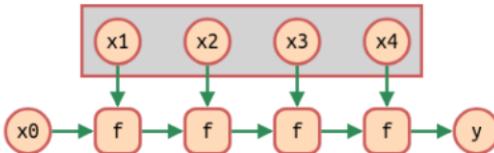
24 / 40

foldl

- Signatura:

foldl :: ($b \rightarrow a \rightarrow b$) $\rightarrow b \rightarrow [a] \rightarrow b$

- Descripció: $\text{foldl } \oplus x_0 xs$ desplega un operador \oplus per l'esquerra, de forma que $\text{foldl } \oplus x_0 [x_1, x_2, \dots, x_n]$ és $((x_0 \oplus x_1) \oplus x_2) \oplus \dots \oplus x_n$.



$y = \text{foldl } f x_0 [x_1, x_2, x_3, x_4]$

- Exemples:

$\lambda> \text{foldl } (+) 0 [3, 2, (-1)]$ -- $((0 + 3) + 2) + (-1)$
👉 4

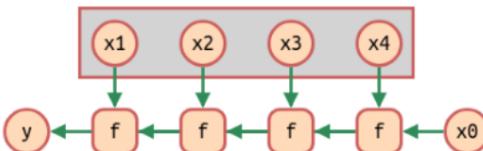
25 / 40

foldr

- Signatura:

foldr :: ($a \rightarrow b \rightarrow b$) $\rightarrow b \rightarrow [a] \rightarrow b$

- Descripció: $\text{foldr } \oplus x_0 xs$ desplega un operador per la dreta, de forma que $\text{foldr } \oplus x_0 [x_1, x_2, \dots, x_n]$ és $x_1 \oplus (x_2 \dots \oplus (x_n \oplus x_0))$.



$y = \text{foldr } f x_0 [x_1, x_2, x_3, x_4]$

- Exemples:

$\lambda> \text{foldr } (+) 0 [3, 2, (-1)]$ -- $3 + ((2 + ((-1) + 0)))$
👉 4

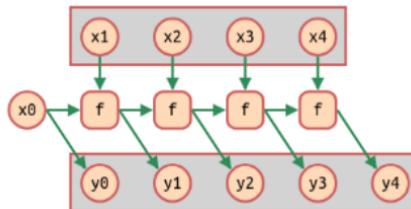
26 / 40

scanl

- Signatura:

scanl :: ($b \rightarrow a \rightarrow b$) $\rightarrow b \rightarrow [a] \rightarrow [b]$

- Descripció: $scanl f x_0 xs$ és com $foldl f x_0 xs$ però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.



$$[y_0, y_1, y_2, y_3, y_4] = scanl f x_0 [x_1, x_2, x_3, x_4]$$

- Exemples:

```
λ> scanl (+) 0 [3, 2, (-1)]  
👉 [0, 3, 5, 4]
```

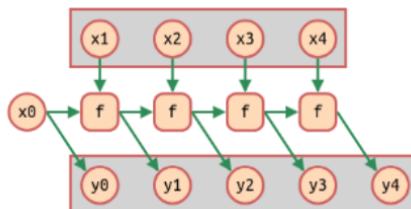
27 / 40

Scanr

- Signatura:

scanr :: ($a \rightarrow b \rightarrow b$) $\rightarrow b \rightarrow [a] \rightarrow [b]$

- Descripció: $scanr f x_0 xs$ és com $foldr f x_0 xs$ però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.



$$[y_0, y_1, y_2, y_3, y_4] = scanr f x_0 [x_1, x_2, x_3, x_4]$$

- Exemples:

```
λ> scanr (+) 0 [3, 2, (-1)]  
👉 [4, 1, -1, 0]
```

28 / 40

Perspectiva

map

- C++

```
vector<X> xs = { ... };

vector<Y> ys;
for (int i = 0; i < xs.size(); ++i) {
    ys.push_back(func(xs[i]));
}
```

- Haskell

```
ys = map func xs
```

29 / 40

Perspectiva

filter

- C++

```
vector<X> xs = { ... };

vector<X> ys;
for (int i = 0; i < xs.size(); ++i) {
    if (pred(xs[i])) {
        ys.push_back(xs[i]);
    }
}
```

- Haskell

```
ys = filter pred xs
```

30 / 40

Perspectiva

foldl

- C++

```
vector<X> xs = { ... };

Y y = zero;
for (int i = 0; i < xs.size(); ++i) {
    y = oper(y, xs[i]);
}
```

- Haskell

```
y = foldl oper zero xs
```

31 / 40

Perspectiva

composició

- Haskell

```
(take 3 . reverse . sort) dades
```

- Shell

```
cat dades | sort | tac | head -3
```

32 / 40

Contingut

- Funcions d'ordre superior
- Funcions d'ordre superior habituals
- Aplicacions
- Exercicis

33 / 40

Diccionaris

Volem definir un TAD Diccionari de Strings a Ints amb valors per defecte usant funcions d'ordre superior.

Interfície

```
type Dict = (String -> Int)      -- Defineix un tipus sinònim a la typedef  
  
create :: Int -> Dict  
search :: Dict -> String -> Int  
insert :: Dict -> String -> Int -> Dict
```

Primera versió

```
type Dict = (String -> Int)  
  
create def = \key -> def  
  
search dict key = dict key  
  
insert dict key value = \x ->  
    if key == x then value  
    else search dict x
```

Segona versió

```
type Dict = (String -> Int)  
  
create = const  
  
search = ($)  
  
insert dict key value x  
    | key == x      = value  
    | otherwise     = dict x
```

34 / 40

Dividir i vèncer

Funció d'ordre superior genèrica `dIv` per l'esquema de dividir i vèncer.

Interfície

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) ->
```

on `a` és el tipus del problema, `b` és el tipus de la solució, i
`dIv trivial directe dividir vèncer x` utilitza:

- `trivial` :: `a -> Bool` per saber si un problema és trivial.
- `directe` :: `a -> b` per solucionar directament un problema trivial.
- `dividir` :: `a -> (a, a)` per dividir un problema no trivial en un parell de subproblems més petits.
- `vèncer` :: `a -> (a, a) -> (b, b) -> b` per, donat un problema no trivial, els seus subproblems i les seves respectives subsolucions, obtenir la solució al problema original.
- `x` :: `a` denota el problema a solucionar.

35 / 40

Dividir i vèncer

Solució

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) ->  
dIv trivial directe dividir vèncer x  
| trivial x      = directe x  
| otherwise       = vèncer x (x1, x2) (y1, y2)  
    where  
        (x1, x2) = dividir x  
        y1 = dIv trivial directe dividir vèncer x1  
        y2 = dIv trivial directe dividir vèncer x2
```

36 / 40

Dividir i vèncer

Solució capturant el context

```
dIv :: (a -> Bool) -> (a -> b) -> (a -> (a, a)) -> (a -> (a, a) -> (b, b) -> b) ->  
dIv trivial directe dividir vèncer = dIv'  
where  
  dIv' x  
    | trivial x = directe x  
    | otherwise = vèncer x (x1, x2) (y1, y2)  
      where  
        (x1, x2) = dividir x  
        y1 = dIv' x1  
        y2 = dIv' x2
```

37 / 40

Dividir i vèncer

Implementació de Quicksort amb Dividir i vèncer

```
qs :: Ord a => [a] -> [a]  
qs = dIv trivial directe dividir vèncer  
where  
  trivial [] = True  
  trivial [_] = True  
  trivial _ = False  
  
directe = id  
  
dividir (x:xs) = (menors, majors)  
  where menors = filter (<= x) xs  
        majors = filter (> x) xs  
  
dividir' (x:xs) = partition (<= x) xs -- equivalent amb funció predefinida  
vèncer (x:_ ) _ (ys1, ys2) = ys1 ++ [x] ++ ys2
```

38 / 40

Contingut

- Funcions d'ordre superior
- Funcions d'ordre superior habituals
- Aplicacions
- Exercicis

39 / 40

Exercicis

1. Feu aquests problemes de Jutge.org:

- P93632 Usage of higher-order functions (1)
- P31745 Usage of higher order functions (2)
- P90677 Definition of higher-order functions (1)
- P71775 Definition of higher-order functions (2)

2. Re-implementeu les funcions habituals sobre llistes.

- Useu `myLength` enlloc de `length` per evitar xocs de noms.
- No useu recursivitat: useu funcions d'ordre superior.

3. Busqueu a [Hoogλe](#) informació sobre aquestes funcions:

- `foldl1`, `foldr1`, `scanl1`, `scanr1`
- `partition`
- `concatMap`
- `zipWith3`
- `mapAccumL`, `mapAccumR`

40 / 40

Llenguatges de Programació

Interludi: aplicació vs composició



Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 9

Contingut

- Aplicació vs composició
- Notació *point-free*
- Exemple amb 2 paràmetres

2 / 9

Aplicació vs composició

Aplicació:

```
($) :: (a -> b) -> a -> b  
-- ($) f x  
--   f   $   x  
-- funció     valor
```

Composició:

```
(.) :: (b -> c) -> (a -> b) -> a -> c  
-- (.) f g x  
--   f   .   g  
-- funció     funció
```

Relació entre l'aplicació i la composició:

```
\x -> f (g x) ≡ \x -> f $ g x ≡ f . g
```

Exemple:

```
appli2 f x = f $ f x  
appli2 (*2) 2 ⚡ 8
```

```
appli2 f = f . f  
appli2 (*2) 2 ⚡ 8
```

3 / 9

Contingut

- Aplicació vs composició
- Notació *point-free*
- Exemple amb 2 paràmetres

4 / 9

Notació *Point-free*

Notació per definir funcions sense explicitar els paràmetres que porta la funció. Normalment les definim utilitzant la composició de funcions.

Exemple:

Definició d'un funció que, donada una llista, ens torni el número de parell que conté.

```
numParells :: Integral a => [a] -> Int
numParells l = length (filter even l)
```

```
-- Point-free
numParells :: Integral a => [a] -> Int
numParells = length . filter even
```

Les dues definicions són completament equivalents:

```
numParells [1..5] ➔ 2
```

5 / 9

Contingut

- Aplicació vs composició
- Notació *point-free*
- Exemple amb 2 paràmetres

6 / 9

Exemple amb 2 paràmetres

Definició d'un funció que, donada una llista i un enter, ens torni el número de vegades que apareix l'enter.

```
numVegades :: Eq a => a -> [a] -> Int
numVegades x l = length $ filter (== x) l

numVegades 3 [3,2,3] ⏪ 2

numVegades x l = (length . filter (== x)) l      -- canviem $ per .

numVegades x = length . filter (== x)           -- treiem l

numVegades x = length . (filter ((==) x))       -- treiem x de (== x)

numVegades x = length . (filter $ (==) x)        -- canviem () per $

numVegades x = length . (filter . (==)) x        -- canviem $ per .
```

7 / 9

Exemple amb 2 paràmetres

En aquest punt tenim:

```
numVegades x = length . (filter . (==)) x    -- necessitem quelcom com (f . g) x
```

Tenim els tipus simplificats i la g:

```
g :: Int -> [Int] -> [Int]
g = filter . (==)

numVegades :: Int -> [Int] -> Int
numVegades = f . g

(.) :: (b -> c) -> (a -> b) -> a -> c
a ≡ Int
b ≡ [Int] -> [Int]
c ≡ Int

length :: [Int] -> Int

f :: ([Int] -> [Int]) -> [Int] -> Int
f ??? length???
```

8 / 9

Exemple amb 2 paràmetres

Solució:

```
f :: ([Int] -> [Int]) -> [Int] -> Int
f = (length .)

f (map id) [2,4,1] ⚡ 3

numVegades x = ((length .) . (filter . (==))) x      -- aplicuem f . g

numVegades = ((length .) . (filter . (==)))           -- treiem x
```

Test:

```
numVegades = (length .)
              (filter . (==))          f = (length .)
                                         g = filter . (==)

numVegades 3 [3,2,3] ⚡ 2          (f . g) 3 [3,2,3] ⚡ 2
```

Llenguatges de Programació

Sessió 3: Llistes infinites



Jordi Petit, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 21

Contingut

- Llistes per comprensió
- Avaluació mandrosa
- Llistes infinites
- Exercicis

2 / 21

Llistes amb rangs

```
λ> [1 .. 10]  
👉 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
λ> [10 .. 1]  
👉 []  
λ> ['E' .. 'J']  
👉 ['E', 'F', 'G', 'H', 'I', 'J']
```

... amb salt

```
λ> [10, 20 .. 100]  
👉 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
λ> [10, 9 .. 1]  
👉 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  
  
λ> [1, 2, 4, 8, 16 .. 256]  
✗ -- no fa miracles
```

... sense final

```
λ> [1..]  
👉 [1, 2, 3, 4, 5, 6, 7, 8, 9, ..... ]  
λ> [1, 3 ..]  
👉 [1, 3, 5, 7, 9, 11, 13, 15, ..... ]
```

3 / 21

Llistes per comprensió

Una llista per comprensió és una construcció per crear, filtrar i combinar llistes.

Sintaxi semblant a la notació matemàtica de construcció de conjunts.

Ternes pitagòriques en matemàtiques: $\{(x,y,z) / 0 < x \leq y \leq z, x^2 + y^2 = z^2\}$

Ternes pitagòriques en Haskell (fins a n):

```
λ> ternes n = [(x, y, z) | x <- [1..n],  
                           y <- [x..n],  
                           z <- [y..n], x*x + y*y == z*z]  
-- gens efficient  
  
λ> ternes 20  
👉 [(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

4 / 21

Llistes per comprensió

Ús bàsic: expressió amb generador (semblant a `map`)

```
[x*x | x <- [1..100]]
```

Filtre (semblant a `map` i `filter`)

```
[x*x | x <- [1..100], capicua x]
```

Múltiples filters

```
[x | x <- [1..100], x `mod` 3 == 0, x `mod` 5 == 0]
```

Múltiples generadors (producte cartesià)

```
[(x, y) | x <- [1..10], y <- [1..10]]
```

Introducció de noms

```
[q | x <- [10..], let q = x*x, let s = show q, s == reverse s]
```

5 / 21

Llistes per comprensió

Compte amb l'ordre

```
[(x, y) | x <- [1..n], y <- [1..m], even x]
[(x, y) | x <- [1..n], even x, y <- [1..m]]
```

Ternes pitagòriques

```
 terres n = [(x, y, z) | x <- [1..n], y <- [x..n], z <- [y..n], x*x + y*y == z*z]
```

```
 terres n = [(x, y, z) | x <- [1..n],
                      y <- [x..n],
                      let z = floor $ sqrt $ fromIntegral $ x*x + y*y,
                      z <= n,
                      x*x + y*y == z*z]
```

6 / 21

Perspectiva

Haskell

```
[ (x, y) | x <- xs, y <- ys, f x == g y, even x]
```

Python

```
[(x, y) for x in xs for y in ys if x.f == y.g and x%2 == 0]
```

SQL

```
SELECT *
FROM xs
JOIN ys
WHERE xs.f = ys.g
AND xs % 2 = 0
```

C++

```
list<pair<X, Y>> l;
for (X x : xs)
    for (Y y : ys)
        if (x.f == y.g and x%2 == 0)
            l.push_back({x, y});
```

7 / 21

Contingut

- Llistes per comprensió
- Avaluació mandrosa
- Llistes infinites
- Exercicis

Avaluació mandrosa

- L'avaluació mandrosa (*lazy*) només avalua el que cal.
- Un *thunk* representa un valor que encara no ha estat avaluat.
- L'avaluació mandrosa no avalua els *thunks* fins que no ho necessita.
- Les expressions es tradueixen en un graf (no un arbre) que és recorregut per obtenir els elements necessaris.
- Això provoca cert indeterminisme en com s'executa.
- Ineficiència(?). Depèn del compilador i depèn del cas.
- Permet tractar estructures potencialment molt grans o "infinites".

9 / 21

Avaluació mandrosa: C++ vs Haskell

```
int f (int x, int y) { return x; }

int main() {
    int a, b;
    cin >> a >> b;
    cout << f(a, a / b);
}
```

$$\begin{aligned} \lambda > f \ x \ y = x \\ \lambda > a = 2 \\ \lambda > b = 0 \\ \lambda > f \ a \ (\text{div } a \ b) \\ \text{👉 } 2 \end{aligned}$$

●: Divisió per zero quan b és zero.

👉 (div a b) no és avaluat.

```
int f (int x, int y) { return x; }
int h (int x)           { for (;;) ; }

int main() {
    int a, b;
    cin >> a >> b;
    cout << f(a, h(b));
}
```

$$\begin{aligned} \lambda > f \ x \ y = x \\ \lambda > h \ x = h \ x \\ \lambda > f \ 3 \ (h \ 0) \\ \text{👉 } 3 \end{aligned}$$

●: Es penja.

👉 h mai és avaluada.

```
if (x != 0 ? 1 / x : 0) { ... }
if (p != nullptr and p->element == x) { ... }
```

●: aquesta és més mandrosa.

10 / 21

Visualització dels *thunks* amb el *ghci*

```
λ> xs = [x + 1 | x <- [1..10]] :: [Int]      λ> y = head $ tail $ tail xs
λ> :sprint xs                                λ> :sprint y
xs = _                                         y = _
λ> null xs                                    λ> :sprint xs
False                                         xs = [2,_,_,_,_,_,_,_,_,_]
λ> :sprint xs                                λ> y
xs = _ : _                                     4
λ> head xs                                    λ> :sprint y
2                                            y = 4
λ> :sprint xs                                λ> :sprint xs
xs = 2 : _                                     xs = [2,_,4,_,_,_,_,_,_,_]
λ> length xs                                 λ> xs
10                                         [2,3,4,5,6,7,8,9,10,11]
λ> :sprint xs                                λ> :sprint xs
xs = [2,_,_,_,_,_,_,_,_,_]                     xs = [2,3,4,5,6,7,8,9,10,11]
```

11 / 21

Avaluació mandrosa pas a pas

Donades les definicions (nats amb recursivitat infinita):

```
nats = 0 : map (+1) nats          (N)      take 0 _ = []
map f (x:xs) = f x : map f xs    (M)      take n (x:xs) = x : take (n-1) xs  (T1)
                                            (T2)
```

Avaluació pas a pas de `take 2 nats`:

expressió	regla
<code>take 2 nats</code>	
<code>take 2 (0 : map (+1) nats)</code>	(N)
<code>0 : take 1 (map (+1) nats)</code>	(T2)
<code>0 : take 1 (map (+1) (0 : map (+1) nats))</code>	(N)
<code>0 : take 1 ((+1) 0 : (map (+1) (map (+1) nats)))</code>	(M)
<code>0 : take 1 (1 : (map (+1) (map (+1) nats)))</code>	(+1)
<code>0 : 1 : take 0 (map (+1) (map (+1) nats))</code>	(M)
<code>0 : 1 : []</code>	(T1)

12 / 21

Contingut

- Llistes per comprensió
- Avaluació mandrosa
- Llistes infinites
- Exercicis

13 / 21

Zeros

Generació de la llista infinita de zeros

```
zeros :: [Int]
-- amb repeat
zeros = repeat 0

-- amb cycle
zeros = cycle [0]

-- amb iterate
zeros = iterate id 0

-- amb recursivitat infinita
zeros = 0 : zeros

-- prova
λ> take 6 zeros
👉 [0, 0, 0, 0, 0, 0]
```

14 / 21

Naturals

Generació de la llista infinita de naturals

```
naturals :: [Int]  
-- amb rangs infinitos  
naturals = [0..]  
  
-- amb iterate  
naturals = iterate (+1) 0  
  
-- amb recursivitat infinita  
naturals = 0 : map (+1) naturals  
  
-- prova  
λ> take 6 naturals  
👉 [0, 1, 2, 3, 4, 5]
```

15 / 21

Factorials

Generació de la llista infinita de factorials

```
factorials :: [Integer]  
factorials = 1:zipWith (*) factorials [1..]  
  
λ> take 10 factorials  
👉 [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]  
  
factorials :: [Integer]  
factorials = scanl (*) 1 [1..]  
  
λ> take 6 $ scanl (*) 1 [1..]  
👉 [1, 1, 2, 6, 24, 120]
```

16 / 21

Fibonacci

Generació de la llista infinita de nombres de Fibonacci

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```
fibs :: [Integer]
fibs = fibs' 0 1
where
  fibs' m n = m : fibs' n (m+n)
```

```
fibs :: [Integer]
fibs = [a+b | (a,b) <- zip (1:fibs) (0:1:fibs)]
```

17 / 21

Primers

Generació dels nombres primers amb el Garbell d'Eratòstenes

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

18 / 21

Avaluació ansiosa

En Haskell es pot forçar cert nivell d'avaluació ansiosa (*eager*) usant l'operador infix `$!`.

`f $! x` evalua primer `x` i després `f x` però només evalua fins que troba un constructor.

19 / 21

Contingut

- Llistes per comprensió
- Avaluació mandrosa
- Llistes infinites
- Exercicis

20 / 21

Exercicis

Feu aquests problemes de Jutge.org:

- [P93588 Usage of comprehension lists](#)
- [P98957 Infinite lists](#)

Llenguatges de Programació

Sessió 4: tipus algebraics i classes



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 44

Contingut

- Tipus
- Tipus algebraics
- Tipus genèrics predefinits
- Classes
- Exercicis

2 / 44

Tipus predefinits

Ja hem vist que existeixen una sèrie de tipus predefinits:

- Tipus simples:
 - Int, Integer, Float, Double
 - Bool
 - Char
- Tipus estructurats:
 - Llistes
 - Tuples
 - Funcions

```
5 :: Integer
True :: Bool
'a' :: Char
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
not :: Bool -> Bool
```

Tots els identificadors de tipus comencem amb majúscula.

3 / 44

Tipus polimòrfics

```
length :: [a] -> Int
map :: (a -> b) -> [a] -> [b]
```

El polimorfisme paramètric és un mecanisme senzill que permet definir funcions (i tipus) que s'escriuen genèricament, sense dependre dels tipus dels objectes sobre els quals s'apliquen.

En Haskell, les variables de tipus poden prendre qualsevol valor i estan quantificades universalment. Per convenció a, b, c, ...

4 / 44

Tipus polimòrfics

Per a utilitzar funcions amb tipus polimòrfics cal que hi hagi una substitució de les variables de tipus que s'adeqüi a l'aplicació que estem fent.

Exemple: `map even [3,6,1]` té tipus `[Bool]` ja que:

- el tipus de `map` és `(a -> b) -> [a] -> [b]`,
- el tipus de `even` és `Int -> Bool`,
- per tant, `a` es pot substituir per `Int` i `b` es pot substituir per `Bool`,
- i el tipus final de l'expressió és `[Bool]`.

Una expressió dóna error de tipus si no existeix una substitució per a les seves variables de tipus.

Exemple: `map not ['b','c']` dóna error de tipus ja que:

- per una banda, `a` hauria de ser `Bool`,
- per altre banda, `a` hauria de ser `Char`.

5 / 44

Tipus sinònims

La construcció `type` permet substituir un tipus (complex) per un nou nom.

Els dos tipus són intercanviables.

```
type Euros = Float

sou :: Persona -> Euros

type Diccionari = String -> Int

crear :: Diccionari
cercar :: Diccionari -> String -> Int
inserir :: Diccionari -> String -> Int -> Diccionari
esborrar :: Diccionari -> String -> Diccionari
```

Els tipus sinònims aporten claredat (però no més seguretat).

💡 Per a més seguretat, mireu `newtype` (no el considerem).

6 / 44

Contingut

- Tipus
- **Tipus algebraics**
- Tipus genèrics predefinits
- Classes
- Exercicis

7 / 44

Tipus enumerats

Els tipus enumerats donen la llista de valors possibles dels objectes d'aquell tipus.

```
data Jugada = Pedra | Paper | Tisores

data Operador
  = Suma
  | Resta
  | Producte
  | Divisio

data Bool = False | True    -- predefinit
```

Els valors enumerats (constructors), han de començar amb majúscula.

Els tipus enumerats es poden desconstruir amb patrons:

```
guanya :: Jugada -> Jugada -> Bool
-- diu si la primera jugada guanya a la segona

guanya Paper Pedra = True
guanya Pedra Tisores = True
guanya Tisores Paper = True
guanya _ _ = False
```

8 / 44

Tipus algebraics

Els tipus algebraics defineixen diversos constructors, cadascun amb zero o més dades associades.

```
data Forma
= Rectangle Float Float          -- alçada, amplada
| Quadrat Float                  -- mida
| Cercle Float                   -- radi
| Punt
```

Les dades es creen especificant el constructor i els seus valors respectius:

```
λ> r = Rectangle 3 4
λ> :type r
👉 r :: Forma

λ> c = Cercle 2.0
λ> :type c
👉 c :: Forma
```

9 / 44

Tipus algebraics

```
data Forma
= Rectangle Float Float          -- alçada, amplada
| Quadrat Float                  -- mida
| Cercle Float                   -- radi
| Punt
```

Els tipus algebraics es poden desconstruir amb patrons:

```
area :: Forma -> Float

area (Rectangle amplada alçada) = amplada * alçada
area (Quadrat mida) = area (Rectangle mida mida)
area (Cercle radi) = pi * radi^2
area Punt = 0
```

```
λ> area (Rectangle 3 4)
👉 12

λ> c = Cercle 2.0
λ> area c
👉 12.566370614359172
```

10 / 44

Tipus algebraics

Per escriure valors algebraics, cal afegir deriving (Show) al final del tipus.
⇒ més endavant veurem què vol dir.

```
data Punt = Punt Int Int
deriving (Show)

data Rectangle = Rectangle Punt Punt
deriving (Show)

λ> p1 = Punt 2 3
λ> p1
👉 Punt 2 3

λ> p2 = Punt 4 6
λ> p2
👉 Punt 4 6

λ> r = Rectangle p1 p2
λ> r
👉 Rectangle (Punt 2 3) (Punt 4 6)
```

11 / 44

Arbres binaris d'enters

Els tipus algebraics també es poden definir recursivament!

```
data Arbin = Buit | Node Int Arbin Arbin
deriving (Show)

λ> a1 = Node 1 Buit Buit
λ> a2 = Node 2 Buit Buit
λ> a3 = Node 3 a1 a2
λ> a4 = Node 4 a3 Buit
λ> a4
👉 Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit

λ> a5 = Node 5 a4 a4           -- I ❤ sharing
λ> a5
👉 Node 5 (Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit)
              (Node 4 (Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)) Buit)
```

Com sempre, la desconstrucció via patrons marca el camí: 🍁

```
alcada :: Arbin -> Int

alcada Buit = 0
alcada (Node _ fe fd) = 1 + max (alcada fe) (alcada fd)
```

12 / 44

Arbres binaris genèrics

Els tipus algebraics també tenen polimorfisme paramètric!

```
data Arbin a = Buit | Node a (Arbin a) (Arbin a)
  deriving (Show)

a1 :: Arbin Int
a1 = Node 3 (Node 1 Buit Buit) (Node 2 Buit Buit)

a2 :: Arbin Forma
a2 = Node (Rectangle 3 4) (Node (Cercle 2) Buit Buit) (Node Punt Buit Buit)

alcada :: Arbin a -> Int
alcada Buit = 0
alcada (Node _ fe fd) = 1 + max (alcada fe) (alcada fd)

preordre :: Arbin a -> [a]
preordre Buit = []
preordre (Node x fe fd) = [x] ++ preordre fe ++ preordre fd
```

13 / 44

Arbres generals genèrics

```
data Argal a = Argal a [Argal a] -- (no hi ha arbre buit en els arbres generals)
  deriving (Show)

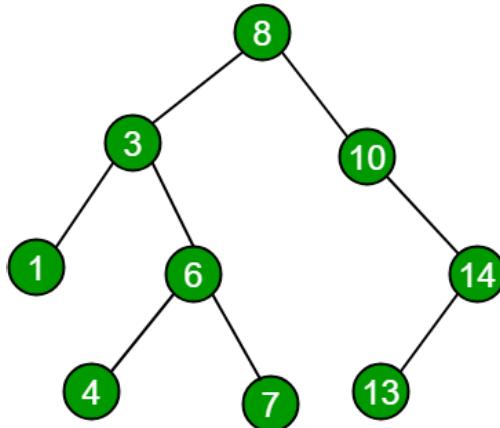
a = Argal 4 [Argal 1 [], Argal 2 [], Argal 3 [Argal 0 []]]

mida :: Argal a -> Int
mida (Argal _ fills) = 1 + sum (map mida fills)

preordre :: Argal a -> [a]
preordre (Argal x fills) = x : concatMap preordre fills
```

14 / 44

Arbres binaris de cerca



```
data Abc a = Buit | Node a (Abc a) (Abc a)           -- arbre binari de cerca

buit      :: Abc a
cerca     :: Ord a => a -> Abc a -> Bool          -- retorna un arbre buit
insereix   :: Ord a => a -> Abc a -> Abc a          -- diu si un abre conté un element
esborra    :: Ord a => a -> Abc a -> Abc a          -- inserció d'un element
                                         -- esborrat d'un element (exercici)
```

15 / 44

Arbres binaris de cerca

```
data Abc a = Buit | Node a (Abc a) (Abc a)           -- arbre binari de cerca

buit :: Abc a                                     -- retorna un arbre buit
buit = Buit

cerca :: Ord a => a -> Abc a -> Bool          -- diu si un abre conté un element
cerca x Buit = False
cerca x (Node k fe fd)
| x < k      = cerca x fe
| x > k      = cerca x fd
| x == k     = True

insereix :: Ord a => a -> Abc a -> Abc a        -- inserció d'un element
insereix x Buit = Node x Buit Buit
insereix x (Node k fe fd)
| x < k      = Node k (insereix x fe) fd
| x > k      = Node k fe (insereix x fd)
| x == k     = Node k fe fd

esborra :: Ord a => a -> Abc a -> Abc a          -- esborrat d'un element (exercici)
```

16 / 44

Expressions booleanes amb variables

```
data ExprBool
= Val Bool
| Var Char
| Not ExprBool
| And ExprBool ExprBool
| Or ExprBool ExprBool
deriving (Show)

type Dict = Char -> Bool

eval :: ExprBool -> Dict -> Bool

eval (Val x) d = x
eval (Var v) d = d v
eval (Not e) d = not $ eval e d
eval (And e1 e2) d = eval e1 d && eval e2 d
eval (Or e1 e2) d = eval e1 d || eval e2 d

e = (And (Or (Val False) (Var 'x')) (Not (And (Var 'y') (Var 'z'))))
d = ('elem' "xz")
eval e d
-- evalua (F V x) & (¬ (y & z)) amb x = z = T i y = F
```

17 / 44

Perspectiva

```
data Expr a
= Val a
| Var String
| Neg (Expr a)
| Sum (Expr a) (Expr a)
| Res (Expr a) (Expr a)
| Mul (Expr a) (Expr a)
| Div (Expr a) (Expr a)
```

Com seria en C++?

18 / 44

Perspectiva

```
template <typename a> class Expr {  
    struct ValData {  
        a x;  
    };  
  
    struct VarData {  
        string v;  
    };  
  
    struct NegData {  
        Node* e;  
    };  
  
    struct OpData {  
        Node* e1;  
        Node* e2;  
    };  
  
    enum Constructor {Val, Var, Neg,  
                      Sum, Res, Mul, Div};  
  
    struct Node {  
        Constructor c;  
        union {  
            ValData val;  
            VarData var;  
            NegData neg;  
            OpData op;  
        };  
    };  
    Node* p; // punter al node amb  
              // l'expressió  
public:  
    Expr ExprVal (const a& x);  
    Expr ExprVar (const string& v);  
    Expr ExprNeg (const Expr& e);  
    Expr ExprSum (const Expr& e1,  
                  const Expr& e2);  
    ...  
};
```

I encara falten les operacions i la gestió de la memòria! 😱👨‍💻

19 / 44

Contingut

- Tipus
- Tipus algebraics
- Tipus genèrics predefinits
- Classes
- Exercicis

20 / 44

Llistes genèriques

```
data Llista a = Buida | a `DavantDe` (Llista a)

l1 = 3 `DavantDe` 2 `DavantDe` 4 `DavantDe` Buida

llargada :: Llista a -> Int

llargada Buida = 0
llargada (cap `DavantDe` cua) = 1 + llargada cua
```

21 / 44

Llistes genèriques

```
data Llista a = Buida | a `DavantDe` (Llista a)

l1 = 3 `DavantDe` 2 `DavantDe` 4 `DavantDe` Buida

llargada :: Llista a -> Int

llargada Buida = 0
llargada (cap `DavantDe` cua) = 1 + llargada cua
```

Les llistes de Haskell són exactament això! (amb una mica de sucre sintàctic
à la dreta)

```
data [a] = [] | a : [a]
```

```
l1 = 3:2:4:[] -- l1 = [3, 2, 4]
```

```
length :: [a] -> Int

length [] = 0
length (x:xs) = 1 + length xs
```

22 / 44

Maybe a

El tipus polimòrfic `Maybe a` està predefinit així:

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

Expressa dues possibilitats:

- la presència d'un valor (de tipus `a` amb el constructor `Just`), o
- la seva absència (amb el constructor buit `Nothing`).

Aplicacions:

- Indicar possibles valor nuls.
- Indicar absència d'un resultat.
- Reportar un error.

Exemples: (busqueu doc a [Hoogλe](#))

```
find :: (a -> Bool) -> [a] -> Maybe a
-- cerca en una llista amb un predicat

lookup :: Eq a => a -> [(a,b)] -> Maybe b
-- cerca en una llista associativa
```

23 / 44

Either a b

El tipus polimòrfic `Either a b` està predefinit així:

```
data Either a b = Left a | Right b
```

Expressa dues possibilitats per un valor:

- un valor de tipus `a` (amb el constructor `Left`), o
- un valor de tipus `b` (amb el constructor `Right`).

Aplicacions:

- Indicar que un valor pot ser, alternativament, de dos tipus.
- Reportar un error. Habitualment:
 - `a` és un `String` i és el diagnòstic de l'error.
 - `b` és del tipus del resultat esperat.
 - Mnemotècnic: `right` vol dir `dreta` i també `correcte`.

Exemple:

```
secDiv :: Float -> Float -> Either String Float
secDiv _ 0 = Left "divisió per zero"
secDiv x y = Right (x / y)
```

24 / 44

Contingut

- Tipus
- Tipus algebraics
- Tipus genèrics predefinits
- Classes
- Exercicis

25 / 44

Classes de tipus

Una classe de tipus (*type class*) és una interfície que defineix un comportament.

Els tipus poden instanciar (implementar seguint la interfície) una o més classes de tipus.

La instació es pot fer

- automàticament pel compilador per a certes classes predefinides, o
- a mà.

Les classes de tipus

- són la forma de tenir sobrecàrrega en Haskell, i
- proporcionen una altra forma de polimorfisme.

⚠️ Les classes de tipus de Haskell no són classes de OOP com a C++ o Java (més aviat són com els interfaces de Java).

26 / 44

La classe Eq

La funció `elem` necessita comparar elements per igualtat:

```
elem :: (Eq a) => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) = x == y || elem x ys
```

La declaració `(Eq a) =>` indica que els tipus `a` sobre els quals es pot aplicar la funció `elem` han de ser instàncies de la classe `Eq`.

La classe predefinida `Eq` dóna operacions d'igualtat i desigualtat:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

I fins i tot ja proporciona definicions per defecte (circulars, què hi farem!):

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

27 / 44

La classe Eq

El nostre tipus `Jugada` (encara) no dóna suport a la classe `Eq`:

```
data Jugada = Pedra | Paper | Tisora

λ> Paper /= Paper
👉 error: "No instance for (Eq Jugada) arising from a use of '/='"

λ> Pedra `elem` [Paper, Pedra, Paper]
👉 error: "No instance for (Eq Jugada) arising from a use of 'elem'"
```

Amb `deriving (Eq)` demanem al compilador que instanciï automàticament la classe `Eq` (usant igualtat estructural):

```
data Jugada = Pedra | Paper | Tisora
deriving (Eq)

λ> Paper /= Paper
👉 False

λ> Pedra `elem` [Paper, Pedra, Paper]
👉 True
```

28 / 44

La classe Eq

Per alguns tipus, la igualtat estructural no és suficient:

```
data Racional = Racional Int Int          -- numerador, denominador
  deriving (Eq)

λ> Racional 3 2 == Racional 6 4
👉 False
```

En aquests casos cal instanciar la classe a mà:

```
instance Eq Racional where
  (Racional n1 d1) == (Racional n2 d2) = n1 * d2 == n2 * d1

λ> Racional 3 2 == Racional 6 4
👉 True

λ> Racional 3 2 /= Racional 6 4
👉 False
```

Només cal definir `==` perquè la definició per defecte de `/=` ja ens convé.

29 / 44

La classe Eq

Per alguns tipus, instanciar una classe també requereix alguna altra classe:

```
data Arbin a = Buit | Node a (Arbin a) (Arbin a)

instance Eq a => Eq (Arbin a) where
  Buit == Buit = True
  (Node x1 fe1 fd1) == (Node x2 fe2 fd2) = x1 == x2 && fe1 == fe2 && fd1 == fd2
  _ == _ = False
```

30 / 44

Informació sobre instàncies

Amb la comanda `:info T` (o `:i T`) de l'interpret es pot veure de quines classes és instància un tipus `T`:

```
λ> :i Racional
data Racional = Racional Int Int
*instance Eq Racional

λ> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
*instance Eq Int
instance Ord Int
instance Show Int
instance Read Int
instance Enum Int
instance Num Int
instance Real Int
instance Bounded Int
instance Integral Int
```

31 / 44

La classe Ord

La classe predefinida `Ord` (que requereix la classe `Eq`) dóna operacions d'ordre:

```
data Ordering = LT | EQ | GT      -- possibles resultats d'una comparació d'ordre

class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<), (≤), (≥), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
    compare x y
        | x == y     = EQ
        | x <= y    = LT
        | otherwise   = GT
    x < y = compare x y == LT
    x > y = compare x y == GT
    x ≤ y = compare x y /= GT
    x ≥ y = compare x y /= LT
```

El mínim que cal per fer la instanciació és definir el `<=` o el `compare`.

Tot i que no es verifica, s'espera que les instàncies d'`Ord` compleixin les lleis:

- Transitivitat: si $x \leq y \ \&\& \ y \leq z$ llavors $x \leq z$.
- Reflexivitat: $x \leq x$.
- Antisimetria: si $x \leq y \ \&\& \ y \leq x$ llavors $x == y$.

32 / 44

La classe Show

La classe predefinida `Show` dóna suport per convertir valors en textos:

```
class Show a where
    show :: a -> String
```

Amb `deriving (Show)`, el compilador la ofereix automàticament (usant sintaxi Haskell):

```
data Racional = Racional Int Int           -- numerador, denominador
    deriving (Eq, Show)

λ> show $ Racional 3 2  ↗ "Racional 3 2"
λ> show $ Racional 6 4  ↗ "Racional 6 4"  ↘
```

Alternativament, per fer la instació a mà només cal definir el `show`:

```
instance Show Racional where
    show (Racional n d) = (show $ div n m) ++ "/" ++ (show $ div d m)
        where m = gcd n d

λ> show $ Racional 3 2  ↗ "3/2"
λ> show $ Racional 6 4  ↗ "3/2"  ↘
```

33 / 44

La classe Read

La classe predefinida `Read` dóna suport per convertir textos en valors:

```
class Read a where
    read :: String -> a
```

Amb `deriving (Read)`, el compilador la ofereix automàticament (usant sintaxi Haskell).

Alternativament, per fer la instació a mà cal definir el `readPrec`, que forma part dels *parsers* interns de Haskell.

Compte: Al usar `read`, sovint cal especificar el tipus de retorn, perquè el compilador sàpiga a quin de tots els `reads` sobrecarregats ens referim:

```
λ> read "38"
λ> (read "38") :: Int  ↗ "Exception: Prelude.read: no parse"
λ> (read "38") :: Integer  ↗ 38
λ> (read "38") :: Float  ↗ 38
λ> (read "38") :: Double  ↗ 38.0
```

34 / 44

La classe Num

La classe predefinida `Num` dóna suport a operadors aritmètics bàsics:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)      :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger         :: Integer -> a

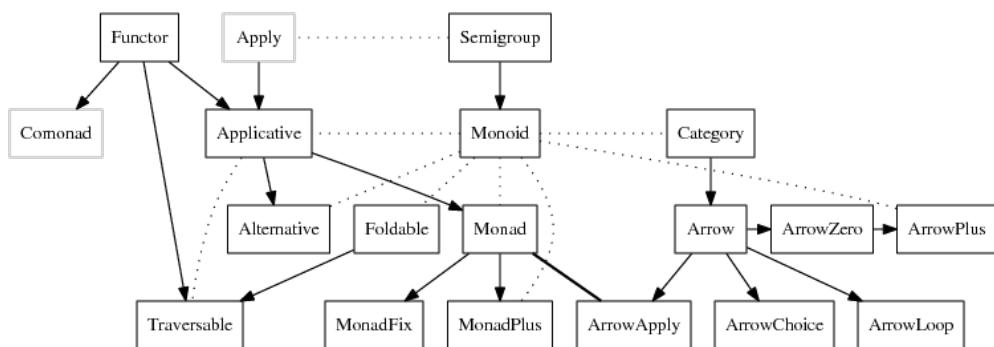
  x - y    = x + negate y
  negate x = 0 -x
```

Per fer la instanciació cal definir totes les operacions menys `negate` o `-`.

Els tipus `Int`, `Integer`, `Float` i `Double` són instàncies de la classe `Num`.

35 / 44

Altres classes predefinides



Font: [Typeclassopedia](#)

36 / 44

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

37 / 44

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

38 / 44

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

39 / 44

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

```
suma :: Num a => [a] -> a
```

✓ el tipus `a` ha de ser instància de `Num`!

40 / 44

Ús de classes en declaracions de tipus

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

Quin és el tipus de `suma`?

```
suma :: [Int] -> Int
```

✗ més general!

```
suma :: [a] -> a
```

✗ el tipus `a` no pot ser qualsevol: ha de tenir l'operació `+`!

```
suma :: Num a => [a] -> a
```

✓ el tipus `a` ha de ser instància de `Num`!

`=>`: condicions sobre les variables de tipus

Haskell és capaç d'inferir tipus i condicions automàticament (més endavant veurem com).

41 / 44

Definició de classes pròpies

Només cal utilitzar la mateixa sintaxi que ja hem vist.

Exemple: Classe per a predicats.

```
class Pred a where
    sat   :: a -> Bool
    unsat :: a -> Bool

    unsat = not . sat
```

Instanciació pels enters:

```
instance Pred Int where
    sat 0 = False
    sat _ = True
```

Instanciació pels arbres binaris:

```
instance Pred a => Pred (Arbin a) where
    sat Buit = True
    sat (Node x fe fd) = sat x && sat fe && sat fd
```

42 / 44

Contingut

- Tipus
- Tipus algebraics
- Tipus genèrics predefinits
- Classes
- [Exercicis](#)

43 / 44

Exercicis

Feu aquests problemes de Jutge.org:

- [P97301 FizzBuzz](#)
- [P37072 Arbre binari](#)
- [P80618 Cua \(1\)](#)

44 / 44

Llenguatges de Programació

Sessió 5: mònades



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 46

Contingut

- [Functors](#)
- [Aplicatius](#)
- [Mònades](#)
- [Entrada/sortida](#)
- [Exercicis](#)

Functors

Ja sabem aplicar funcions:

$\lambda> (+3) 2$

👉 5

Però...

$\lambda> (+3) (\text{Just } 2)$

✗

En aquest cas, podem fer servir `fmap`!

$\lambda> \text{fmap } (+3) (\text{Just } 2)$

👉 `Just 5`
👉 `Nothing`

I també funciona amb `Either`, llistes, tuples i funcions:

$\lambda> \text{fmap } (+3) (\text{Right } 2)$

👉 `Right 5`
👉 `Left "err"`

$\lambda> \text{fmap } (+3) [1, 2, 3]$

👉 `[4, 5, 6]` -- igual que `map`

$\lambda> \text{fmap } (+3) (1, 2)$

👉 `(1, 5)` -- perquè `(,)` és un tipus

$\lambda> (\text{fmap } (*2) (+1)) 3$

👉 `8` -- igual que `(.)`

3 / 46

Functors

`fmap` aplica una funció als elements d'un contenidor genèric `f` a retornant un contenidor del mateix tipus.

`fmap` és una funció de les instàncies de la classe `Functor`:

```
 $\lambda> :type \text{fmap}$ 
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

On

```
 $\lambda> :info Functor$ 
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

4 / 46

Maybe és functor

El tipus Maybe és instància de Functor:

```
λ> :info Maybe
data Maybe a = Nothing | Just a
instance Ord a => Ord (Maybe a)
instance Eq a => Eq (Maybe a)
instance Applicative Maybe
*instance Functor Maybe
instance Monad Maybe
:
```

Concretament,

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

5 / 46

Aplicació

Consulta a una BD:

- Llenguatge sense Maybe:

```
post = Posts.find(1234)
if post is None:
    return None
else:
    return post.title
```

- En Haskell:

```
fmap getPostTitle (findPost 1234)
```

o també:

```
getPostTitle `fmap` findPost 1234
```

o millor (<\$> és l'operador infix per a `fmap`): (es llegeix *fmap*)

```
getPostTitle <$> findPost 1234
```

6 / 46

Either a és functor

El tipus Either a és instància de Functor:

```
instance Functor (Either a) where
    fmap f (Left x) = Left x
    fmap f (Right x) = Right (f x)
```

Fixeu-vos que Either té dos paràmetres:

- el tipus Either a és la instància de Functor,
- el tipus Either no.

7 / 46

Les llistes són functors

El tipus [] (llista) és instància de Functor:

```
instance Functor [] where
    fmap = map           -- potser és al revés, poc importa
```

8 / 46

Les funcions són functors

Les funcions també són instàncies de Functor:

```
instance Functor ((->) r) where
    fmap = (.)
```

Exemple:

```
λ> (*3) <$> (+2) <$> Just 1      ⚡ Just 9
λ> (*3) <$> (+2) <$> Nothing     ⚡ Nothing
```

9 / 46

Lleis dels functors

Les instàncies de functors han de tenir aquestes propietats:

1. Identitat: `fmap id ≡ id`.
2. Composició: `fmap (g1 . g2) ≡ fmap g1 . fmap g2`.

Nota: Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

Exercici: Comproveu que `Maybe`, `Either`, `a`, `[]`, `(,)` i `(->)` compleixen les lleis dels functors.

10 / 46

Arbres binaris com a functors

Instanciació pròpia dels functors pels arbres binaris:

```
data Arbin a
= Buit
| Node a (Arbin a) (Arbin a)
deriving (Show)

instance Functor (Arbin) where
    fmap f Buit = Buit
    fmap f (Node x fe fd) = Node (f x) (fmap f fe) (fmap f fd)

a = Node 3 Buit (Node 2 (Node 1 Buit Buit) (Node 1 Buit Buit))

λ> fmap (*2) a
👉 Node 6 Buit (Node 4 (Node 2 Buit Buit) (Node 2 Buit Buit))

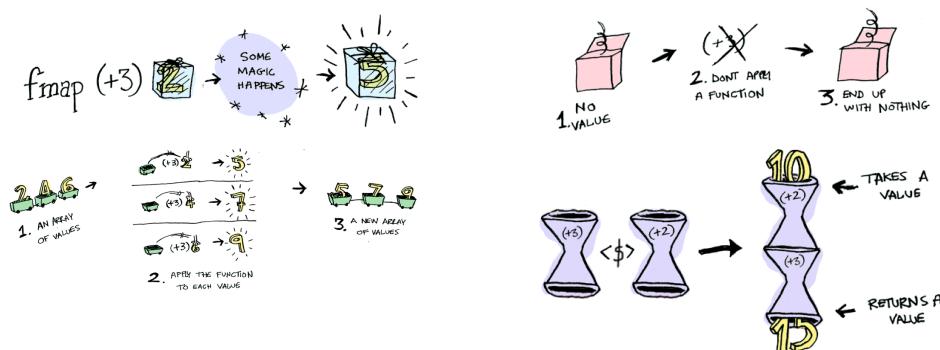
λ> fmap even a
👉 Node False Buit (Node True (Node False Buit Buit) (Node False Buit Buit))
```

Exercici: Comproveu que `Arbin` compleix les lleis dels functors.

11 / 46

Sumari

La classe `Functor` capture la idea de tipus contenidor genèric al qual es pot aplicar una funció als seus elements per canviar el seu contingut (però no el contenidor).



Dibuixos: adit.io

12 / 46

Contingut

- Functors
- **Aplicatius**
- Mònades
- Entrada/sortida
- Exercicis

13 / 46

Aplicatius

Ja sabem aplicar funcions:

$\lambda> (+3) 2$

👉 5

I ho sabem fer sobre contenidors:

$\lambda> fmap (+3) (Just 2)$

👉 Just 5

Però què passa si la funció és en un contenidor?

$\lambda> (Just (+3)) (Just 2)$ ❌

En aquest cas, podem fer servir `<*>!` (es llegeix *app*)

$\lambda> Just (+3) <*> Just 2$	👉 Just 5
$\lambda> Just (+3) <*> Nothing$	👉 Nothing
$\lambda> Nothing <*> Just (+3)$	👉 Nothing
$\lambda> Nothing <*> Nothing$	👉 Nothing
$\lambda> Right (+3) <*> Right 2$	👉 Right 5
$\lambda> Right (+3) <*> Left "err"$	👉 Left "err"
$\lambda> Left "err" <*> Right 2$	👉 Left "err"
$\lambda> Left "err1" <*> Left "err2"$	👉 Left "err1 "
$\lambda> [(*2), (+2)] <*> [1, 2, 3]$	👉 [2, 4, 6, 3, 4, 5]

14 / 46

Aplicatius

L'operador `<*>` és una operació de la classe `Applicative` (que també ha de ser functor):

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> (f a -> f b)
  pure :: a -> f a
```

- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor. Els contenidors són genèrics i del mateix tipus.
- `pure` construeix un contenidor amb un valor.

15 / 46

Lleis dels aplicatius

Les instàncies d'aplicatius han de tenir aquestes propietats:

1. Identitat:

$$\text{pure } \text{id} \text{ } <*> \text{ } v \equiv v.$$

2. Homomorfisme:

$$\text{pure } f \text{ } <*> \text{ } \text{pure } x \equiv \text{pure } (f x).$$

3. Intercanvi:

$$u \text{ } <*> \text{ } \text{pure } y \equiv \text{pure } (\$ y) \text{ } <*> u.$$

4. Composició:

$$u \text{ } <*> \text{ } (v \text{ } <*> \text{ } w) \equiv \text{pure } (.) \text{ } <*> \text{ } u \text{ } <*> \text{ } v \text{ } <*> \text{ } w.$$

5. Relació amb el functor:

$$\text{fmap } g \text{ } x \equiv \text{pure } g \text{ } <*> \text{ } x.$$

16 / 46

Instanciacions d'aplicatius

Maybe és aplicatiu:

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    Just f <*> x = fmap f x
```

Either a és aplicatiu:

```
instance Applicative (Either a) where
    pure = Right
    Left x <*> _ = Left x
    Right f <*> x = fmap f x
```

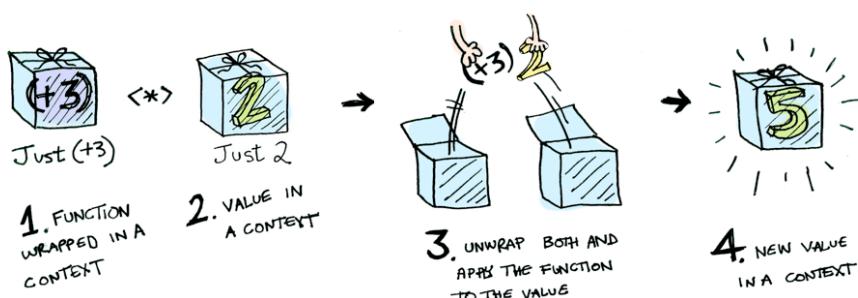
Exercici: Instancieu les llistes com a aplicatius. Hi ha dues formes de fer-ho.

17 / 46

Sumari

Els aplicatius permeten aplicar funcions dins d'un contenidor a objectes dins del mateix contenidor.

- `pure` construeix un contenidor amb un valor.
- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor:



Dibuixos: [adit.io](#)

18 / 46

Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

19 / 46

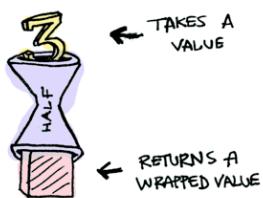
Mònades

Considerem que `meitat` és una funció que només té sentit sobre parells:

```
meitat :: Int -> Maybe Int  
  
meitat x  
| even x    = Just (div x 2)  
| otherwise = Nothing
```

Podem veure la funció així: Donat un valor, retorna un valor empaquetat.

Però llavors no li podem ficar valors empaquetats!



Mònades

Cal una funció que desempaqueti, apliqui meitat i deixi empaquetat.

Aquesta funció es diu `>>=` (es llegeix *bind*)

```
λ> Just 40 >>= meitat  ⚡ Just 20
λ> Just 31 >>= meitat  ⚡ Nothing
λ> Nothing >>= meitat  ⚡ Nothing

λ> Just 20 >>= meitat >>= meitat  ⚡ Just 5
λ> Just 20 >>= meitat >>= meitat  ⚡ Nothing
```

L'operador `>>=` és una operació de la classe Monad:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  -- i més coses
```

El tipus Maybe és instància de Monad:

```
instance Monad Maybe where
  Nothing >>= f    =  Nothing
  Just x   >>= f    =  f x
```

21 / 46

Mònades

De fet, les mònades tenen tres operacions:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b

  r >> k    =    r >>= (\_ -> k)
```

- `return` empaqueta.
- `>>=` desempaqueteta, aplica i empaqueta.
- `>>` és purament estètica.

22 / 46

Instanciacions

Els tipus `Maybe`, `Either a i []` són instàncies de `Monad`:

```
instance Monad Maybe where
    return     = Just
    Nothing >>= f = Nothing
    Just x >>= f = f x

instance Monad (Either a) where
    return     = Right
    Left x >>= f = Left x
    Right x >>= f = f x

instance Monad [] where
    return x      = [x]
    xs >>= f     = concatMap f xs
```

23 / 46

Lleis de les mònades

Les instàncies de mònades han de tenir aquestes propietats:

1. Identitat per l'esquerra:

```
return x >>= f ≡ f x.
```

2. Identitat per la dreta:

```
m >>= return ≡ m.
```

3. Associativitat:

```
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g).
```

Nota: Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

Exercici: Comproveu que `Maybe`, `Either a i []` compleixen les lleis de les mònades.

24 / 46

Notació do

La notació **do** és sucre sintàctic per facilitar l'ús de les mònades.
⇒ Amb do, codi funcional *sempbla* codi imperatiu amb assignacions.

Els còmputs es poden seqüenciar:

do { e1 ; e2 }

≡

do

e1

e2

≡

e1 >> e2

I amb <- extreure el seus resultats:

do { x <- e1 ; e2 }

≡

do

x <- e1

e2

≡

e1 >>= \x -> e2

≡

e1 >>= _ -> e2

25 / 46

Notació do: Exemple

Tenim llistes associatives amb informació sobre propietaris de cotxes, les seves matrícules, els seus models i les seves etiquetes d'emissions:

```
data Model = Seat127 | TeslaS3 | NissanLeaf | ToyotaHybrid deriving (Eq, Show)
data Etiqueta = Eco | B | C | Cap deriving (Eq, Show)
matricules = [("Joan", 6524), ("Pere", 6332), ("Anna", 5313), ("Laia", 9999)]
models = [(6524, NissanLeaf), (6332, Seat127), (5313, TeslaS3), (7572, ToyotaHybrid)]
etiquetes = [(Seat127, Cap), (TeslaS3, Eco), (NissanLeaf, Eco), (ToyotaHybrid, B)]
```

Donat un nom de propietari, volem saber quina és la seva etiqueta d'emissions:

```
etiqueta :: String -> Maybe Etiqueta
```

És Maybe perquè, potser el propietari no existeix, o no tenim la seva matrícula, o no tenim el seu model, o no tenim la seva etiqueta...

26 / 46

Notació do: Exemple

Ens anirà bé usar aquesta funció predefinida de cerca:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Solució amb case: 🍔

```
etiqueta nom =  
  case lookup nom matricules of  
    Nothing -> Nothing  
    Just mat -> case lookup mat models of  
      Nothing -> Nothing  
      Just mod -> lookup mod etiquetes
```

Solució amb notació do: 💜

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

27 / 46

Notació do: Exemple

Amb notació do:

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

Transformació de notació do a funcional:

```
etiqueta nom =  
  lookup nom matricules >>= \mat -> lookup mat models >>= \mod -> lookup mod etiquetes
```

Amb un format diferent queda clara l'equivalència: 😊

```
etiqueta nom =  
  lookup nom matricules >>= \mat ->  
  lookup mat models >>= \mod ->  
  lookup mod etiquetes
```

28 / 46

Funcions predefinides per a mònades

Moltes funcions predefinides tenen una extensió per la classe Monad:

- `mapM`, `filterM`, `foldM`, `zipWithM`, ...

També disposem d'operacions per extender (*lift*) operacions per treballar amb elements de la classe Monad. S'han d'importar:

```
import Control.Monad
liftM  :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

Per exemple, podem crear una funció per suma Maybes:

```
sumaMaybes :: Num a => Maybe a -> Maybe a -> Maybe a
sumaMaybes = liftM2 (+)
```

```
λ> sumaMaybes (Just 3) (Just 4) ➔ Just 7
```

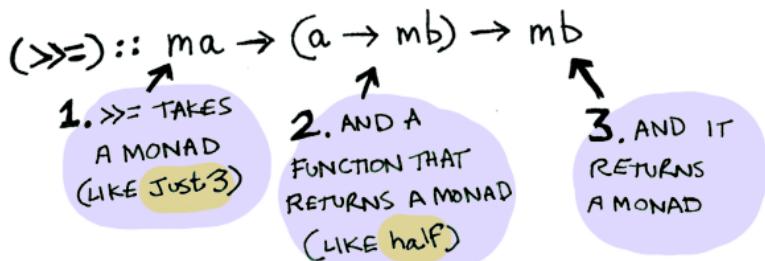
O fer-ho directament:

```
λ> liftM2 (+) (Just 3) (Just 4) ➔ Just 7
```

29 / 46

Sumari (1)

- Les mònades permeten aplicar una funció que retorna un valor en un contenidor a un valor en un contenidor.



Dibuixos: adit.io

- Molts tipus predefinits són instàncies de mònades.
- La notació do simplifica l'ús de les mònades.

30 / 46

Sumari (2)

- Aplicacions:
 - IO
 - Parsers
 - Logging
 - Estat mutable
 - No determinisme
 - Paral·lelisme
- Lectura recomanada: [Monads for functional programming](#) de P. Wadler.

31 / 46

Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

32 / 46

Entrada/Sortida

L'entrada/sortida en Haskell es basa en una mònada:

- El programa principal és `main :: IO ()`
- S'usa el constructor de tipus `IO` per gestionar l'entrada/sortida.
- `IO` és instància de Monad.
- Es sol usar amb notació `do`.

Algunes operacions bàsiques:

```
getChar    :: IO Char          -- obté següent caràcter
getLine    :: IO String        -- obté següent línia
getContents :: IO String       -- obté tota l'entrada

putChar    :: Char -> IO ()   -- escriu un caràcter
putStr     :: String -> IO ()  -- escriu un text
putStrLn   :: String -> IO ()  -- escriu un text i un salt de línia
print      :: Show a => a -> IO () -- escriu qualsevol showable

readFile   :: FilePath -> IO String    -- getContents d'un arxiu
writeFile  :: FilePath -> String -> IO ()  -- operació inversa
```

`()` és una tupla de zero camps i `()` és l'únic valor de tipus `()`.
(\Rightarrow `void` de C).

33 / 46

Hello world!

```
main = do
    putStrLn "Com et dius?"
    nom <- getLine
    putStrLn $ "Hola " ++ nom ++ "!"
```

Compilació i execució:

```
*> ghc programa.hs
[1 of 1] Compiling Main           ( programa.hs, programa.o )
Linking programa ...

*> ./programa
Com et dius?
*Jordi
Hola Jordi!
```

34 / 46

Del revés

```
main = do
    x <- getLine
    let y = reverse x
    putStrLn x
    putStrLn y
```

Compilació i execució:

```
*> ghc programa.hs
[1 of 1] Compiling Main             ( programa.hs, programa.o )
Linking programa ...

*> ./programa
*GAT
GAT
TAG
```

35 / 46

Exemple

Llegir seqüència de línies acabades en * i escriure cadascuna del revés:

```
main = do
    line <- getLine
    if line /= "*" then do
        putStrLn $ reverse line
        main
    else
        return ()
```

36 / 46

Exemple

Llegir seqüència de línies i escriure cadascuna del revés:

```
main = do
    contents <- getContents
    mapM_ (putStrLn . reverse) (lines contents)
```

L'E/S també és *lazy*, no cal preocupar-se perquè l'entrada sigui massa llarga.

37 / 46

where en notació do

Degut a la definició del `>>=`, el `where` pot donar problemes:

```
main = do
    x <- getLine
    print f
    where f = factorial (read x)

X error: Variable not in scope: x :: String
```

Si ho escrivim amb `>>=`, tenim

```
main = getLine >>= \x -> print f
where f = factorial (read x)
```

que no pot ser, ja que a les definicions del `where` no podem usar la variable abstracta `x`.

38 / 46

let en notació do

Amb el do cal usar el let (sense in):

```
main = do
    x <- getLine
    let f = factorial (read x)
    print f
```

Alternativament (més lleig):

```
main = do
    x <- getLine
    f <- return $ factorial (read x)
    print f
```

39 / 46

Intuïció sobre la mònada IO

Entrada/sortida com funcions que modifiquen el món: món1 \rightsquigarrow món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

Exemple: Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món

myGetChar :: World -> (World, Char)

myPutChar :: Char -> World -> (World, ())

myMain :: World -> (World, ())

myMain w0 = let (w1, c1) = myGetChar w0
            (w2, c2) = myGetChar w1
            (w3, ()) = myPutChar c1 w1
            (w4, ()) = myPutChar c2 w3
            in (w4, ())
```

(1) Passant el relleu.

40 / 46

Intuïció sobre la mònada IO

Entrada/sortida com funcions que modifiquen el món: món1 \rightsquigarrow món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

Exemple: Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món           type IO a = World -> (World, a)
myGetChar :: World -> (World, Char)             getChar :: IO Char
myPutChar :: Char -> World -> (World, ())      putChar :: Char -> IO ()
myMain :: World -> (World, ())
myMain w0 = let (w1, c1) = myGetChar w0
            (w2, c2) = myGetChar w1
            (w3, ()) = myPutChar c1 w1
            (w4, ()) = myPutChar c2 w2
            in (w4, ())
main = getChar >>= \c1 ->
        getChar >>= \c2 ->
        putChar c1 >>
        putChar c2
```

(1) Passant el relleu.

(2) Fent que IO sigui instància de Monad.

41 / 46

Intuïció sobre la mònada IO

Entrada/sortida com funcions que modifiquen el món: món1 \rightsquigarrow món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

Exemple: Llegir i escriure dos caràcters.

```
type IO a = World -> (World -> a)
getChar :: IO Char
putChar :: Char -> IO ()
main :: IO ()
main = do
        c1 <- getChar
        c2 <- getChar
        putChar c1
        putChar c2
```

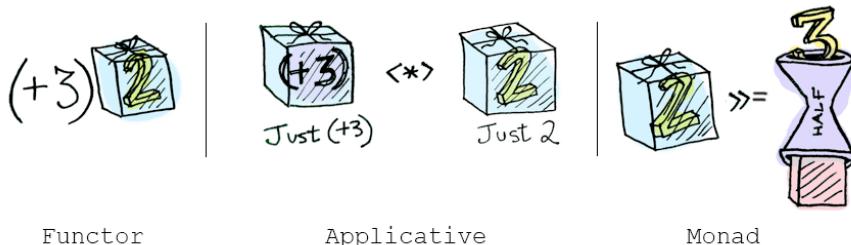
(2) Fent que IO sigui instància de Monad.

(3) Usant notació do.

42 / 46

Sumari

- Hem vist tres classes predefinides molt importants en Haskell: Functors, Aplicatius, Mònades.



- Molts tipus predefinits són instàncies d'aquestes classes: `Maybe`, `Either`, llistes, tuples, funcions, `IO`, ...
- La notació `do` simplifica l'ús de les mònades.
- La classe `IO` permet disposar d'entrada/sortida en un llenguatge funcional pur.

43 / 46

Final

L'estat d'un programa descriu tota la informació que no és local a una funció en particular. Això inclou:

- variables globals
- entrada
- sortida

Pensar sobre un programa amb estat és difícil perquè:

- L'estat perviu d'una crida d'una funció a una altra.
- L'estat és a l'abast de totes les funcions.
- L'estat és mutable.
- L'estat canvia en el temps.
- Cap funció és responsable de l'estat.

Estat: 😞

Sense estat: ❤️

Les mònades no eliminan la noció d'estat en un programa, però eliminan la necessitat de mencionar-lo.

44 / 46

Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- [Exercicis](#)

45 / 46

Exercicis

Feu aquests problemes de Jutge.org:

- Functors, aplicatius i mònades:
 - [P70540](#) Expressions
 - [P50086](#) Cua 2
 - [P58738](#) Arbres amb talla
- Entrada/sortida:
 - [P87974](#) Hola / Adéu
 - [P87082](#) Índex massa corporal

46 / 46

Llenguatges de Programació

Sessió 6: més mònades



Jordi Petit, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 29

Contingut

- Mònade llista
- Mònade estat
- Combinació de mònades

2 / 29

Instanciació de llistes com a mònades

Recordatori de la classe Monad:

```
class Monad m where
  (">>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Les llistes d'a (tipus [a]) instancien les mònades d'aquesta forma:

```
instance Monad [] where
  xs >>= f = concat (map f xs)           -- = concatMap f xs
  return x = [x]
```

Exemple: $f x = [x, 2*x]$ i $xs = [1, 2, 3]$.

```
xs >>= f =
  = concat (map f xs)
  = concat (map (\x -> [x, 2*x]) [1, 2, 3])
  = concat ([[1, 2], [2, 4], [3, 6]])
  = [1, 2, 2, 4, 3, 6]                      -- definició de >>=
                                                -- definició de f i xs
                                                -- aplicació de map
                                                -- aplicació de concat
```

3 / 29

Exemple: salts de cavall

Tenim un cavall que es mou en un tauler d'escacs 8×8 .

Les posicions són parells d'enters entre 1 i 8:

```
type Pos = (Int, Int)

dins :: Pos -> Bool
dins (x, y) = dins' x && dins' y    where dins' i = 1 <= i && i <= 8
```

Donada una posició, moviments retorna la llista de posicions a on pot anar un cavall:

```
moviments :: Pos -> [Pos]
moviments (f, c) =
  filter dins [
    (f + 2, c - 1), (f + 2, c + 1), (f - 2, c - 1), (f - 2, c + 1),
    (f + 1, c - 2), (f + 1, c + 2), (f - 1, c - 2), (f - 1, c + 2)]
```

La funció potAnar3, donada una posició inicial p dins del tauler i una posició final q, diu si un cavall pot anar de p a q en (exactament) tres salts:

```
potAnar3 :: Pos -> Pos -> Bool
potAnar3 p q = q `elem` destins
  where destins = (moviments p >>= moviments >>= moviments)
```

4 / 29

Llistes per comprensió

La notació de les llistes per comprensió és sucre sintàctic sobre la notació do.

L'expressió

```
[f x y | x <- xs, y <- ys]
```

és equivalent a

do

```
x <- xs  
y <- ys  
return (f x y)
```

5 / 29

Llistes per comprensió

Exemple:

```
[(x,y) | x <- "abc", y <- [1, 2, 3]]
```

==>

do

```
x <- "abc"  
y <- [1, 2, 3]  
return (x,y)
```

==>

```
"abc" >>= \x ->  
[1, 2, 3] >>= \y ->  
return (x,y)
```

6 / 29

Filtres

Per poder disposar de filters en les llistes de comprensió, les mònades de Haskell també tenen una operació guard:

```
class Monad m where
  (">>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  guard :: Bool -> m ()
```

Els filters en les llistes de comprensió són nou sucre sintàctic sobre guard:

```
[f x y | x <- xs, p x, y <- ys]
```

és equivalent a

```
do
  x <- xs
  guard (p x)
  y <- ys
  return (f x y)
```

Nota: Aquesta explicació és una simplificació de la realitat.

7 / 29

Filtres

Exemple: Tripletes pitagòriques

```
[(x, y, z) | x <- [1..n], y <- [x..n], z <- [y..n], x*x + y*y == z*z]
```

==>

```
do
  x <- [1..n]
  y <- [x..n]
  z <- [y..n]
  guard (x*x + y*y == z*z)
  return (x,y,z)
```

==>

```
[1..n] >>= \x ->
  [x..n] >>= \y ->
    [y..n] >>= \z ->
      guard (x*x + y*y == z*z) >>
        return (x,y,z)
```

(feu `import Control.Monad` per tenir `guard`)

8 / 29

Filtres

Implementació de guard:

```
guard True  =  return ()  
guard False =  []
```

0

```
guard b    =  if b then [()] else []
```

Exemple: Comprovar

```
[x | x <- [1,2,3], odd x]  ↗ [1, 3]
```

9 / 29

Filtres

```
[x | x <- [1,2,3], odd x]
```

```
👉 do  
  x <- [1,2,3]  
  guard (odd x)  
  return x  
  
👉 [1,2,3] >= (\x -> guard (odd x) >> return x)  
👉 [1,2,3] >= (\x -> guard (odd x) >> \_ -> return x)  
👉 concat (map (\x -> guard (odd x) >> \_ -> return x) [1,2,3])  
👉 concat (map (\x -> guard (odd x) >> \_ -> [x]) [1,2,3])  
👉 concat [  
  (\x -> guard (odd x) >> \_ -> [x]) 1,  
  (\x -> guard (odd x) >> \_ -> [x]) 2,  
  (\x -> guard (odd x) >> \_ -> [x]) 3  
]  
👉 concat [  
  guard (odd 1) >> \_ -> [1],  
  guard (odd 2) >> \_ -> [2],  
  guard (odd 3) >> \_ -> [3]  
]
```

10 / 29

Filtres

```
👉 concat [
    guard True  >>= \_ -> [1],
    guard False >>= \_ -> [2],
    guard True   >>= \_ -> [3]
]

👉 concat [
    [() ] >>= \_ -> [1],
    [__] >>= \_ -> [2],
    [() ] >>= \_ -> [3]
]

👉 concat [
    concat (map (\_ -> [1]) [()]),
    concat (map (\_ -> [2]) []),
    concat (map (\_ -> [3]) [()])
]

👉 concat [ concat [[1]], concat [ ], concat [[3]] ]
👉 concat [ [1], [ ], [3] ]
👉 [1, 3]
😊
```

11 / 29

Lets en llistes de comprensió

Per desensucrar els lets dins de llistes per comprensió, només cal posar-los dins la notació do, que ja té lets.

Exemple:

```
[(x, y, z) |
 x <- [1..n],
 y <- [x..n],
 let z = isqrt (x*x + y*y),
 z <= n,
 x*x + y*y == z*z
]
```

Equival a

```
do
  x <- [1..n]
  y <- [x..n]
  let z = isqrt (x*x + y*y)
  guard (z <= n)
  guard (x*x + y*y == z*z)
  return (x,y,z)
```

12 / 29

Sumari

- Les llistes són mònades (`>>=` és `concatMap`).
- Les llistes per comprensió són sucre sintàctic.
- Cal ampliar les mònades amb `guard` per implementar filtres.

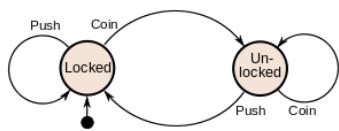
13 / 29

Contingut

- Mònade llista
- Mònade estat
- Combinació de mònades

14 / 29

Exemple: torn d'entrada en C++



```
FSM torn;
torn.getEstat()   ↗ Locked
torn.coinTrigger() ↗ Thank
torn.getEstat()   ↗ Unlocked
torn.pushTrigger() ↗ Open
torn.getEstat()   ↗ Locked
```

```
class FSM {
private:
    string estat;
public:
    FSM() {
        estat = "Locked";
    }
    string getEstat() {
        return estat;
    }
    string coinTrigger() {
        estat = "Unlocked";
        return "Thank";
    }
    string pushTrigger() {
        if (estat == "Locked")
            return "Error";
        estat = "Locked";
        return "Open";
    }
};
```

Fonts: [wikipedia](#) i [wikibooks](#)

15 / 29

Exemple: torn d'entrada en Haskell

Codi

```
data TurnstileState = Locked | Unlocked
deriving (Eq, Show)

data TurnstileOutput = Thank | Open | Error
deriving (Eq, Show)

coinTrigger :: TurnstileState -> (TurnstileOutput, TurnstileState)
coinTrigger _ = (Thank, Unlocked)

pushTrigger :: TurnstileState -> (TurnstileOutput, TurnstileState)
pushTrigger Locked = (Error, Locked)
pushTrigger Unlocked = (Open, Locked)
```

Ús

```
pushTrigger Locked   ↗ (Error,Locked)
coinTrigger Locked   ↗ (Thank,Unlocked)
pushTrigger Unlocked ↗ (Open,Locked)
coinTrigger Unlocked ↗ (Thank,Unlocked)
```

* font: [wikibooks](#)

16 / 29

Exemple: torn d'entrada

En seqüència

```
triggers :: TurnstileState -> ([TurnstileOutput], TurnstileState)
triggers s0 =
  let (a1, s1) = coinTrigger s0
      (a2, s2) = pushTrigger s1
      (a3, s3) = pushTrigger s2
  in ([a1, a2, a3], s3)

triggers Locked ⚡ ([Thank, Open, Error], Locked)
```

Aniria bé tenir l'operador `>>=` per encadenar els *triggers*.

* font: [wikibooks](#)

17 / 29

Mònada State

La mònada *State* encapsula una estructura de dades en forma d'estat.

El tipus *State*

- representa funcions d'estat a tuples (valor, estat):

```
s -> (a, s)
```

- té funcions per aplicar funcions d'estat:

```
runState :: State s a -> s -> (a, s)
runState coinTrigger Locked -- aplica una funció d'estat
```

- és instància de `monad`
- té l'operador `>>=` per enllaçar funcions d'estat

18 / 29

Exemple: torn d'entrada I

Data

```
import Control.Monad
import Control.Monad.State

data TurnstileState = Locked | Unlocked
    deriving (Eq, Show)

data TurnstileOutput = Thank | Open | Error
    deriving (Eq, Show)
```

Funció `coin`

```
coinTrigger :: State TurnstileState TurnstileOutput
coinTrigger = do
    put Unlocked -- nou estat
    return Thank -- valor de tornada
```

Ús amb `runState`

```
runState coinTrigger Locked  ↪ (Thank,Unlocked)
runState coinTrigger Unlocked ↪ (Thank,Unlocked)
```

19 / 29

Exemple: torn d'entrada II

Funció `push`

```
pushTrigger :: State TurnstileState TurnstileOutput
pushTrigger = do
    s <- get -- obté l'estat
    if s == Locked
    then return Error -- valor de tornada
    else do
        put Locked -- nou estat
        return Open -- valor de tornada
```

Ús amb `runState`

```
runState pushTrigger Locked  ↪ (Error,Locked)
runState pushTrigger Unlocked ↪ (Open,Locked)
```

20 / 29

Enllaç i seqüència

Enllaç de *triggers*

```
runState (coinTrigger >= \x -> pushTrigger >= \y -> return [x,y]) Locked
```

👉 ([**Thank**,**Open**],Locked)

```
runState (sequence [coinTrigger, pushTrigger]) Locked
```

👉 ([**Thank**,**Open**],Locked)

Exemple més llarg

```
runState (sequence  
[coinTrigger, pushTrigger, pushTrigger, coinTrigger, pushTrigger])  
Locked
```

👉 ([**Thank**,**Open**,**Error**,**Thank**,**Open**],Locked)

21 / 29

Funcions de la mònada State

Funcions de treball

- **get:** obté l'estat

```
get :: MonadState s m => m s
```

- **put:** modifica l'estat

```
put :: MonadState s m => s -> m ()
```

- **return:** retorna el valor

```
return :: Monad m => a -> m a
```

Funcions de crida

- **runState:** crida a una funció i torna el valor i el nou estat
`runState coinTrigger Locked` 👉 (**Thank**,**Unlocked**)

- **evalState:** crida a una funció i torna el valor
`evalState coinTrigger Locked` 👉 **Thank**

- **execState:** crida a una funció i torna el nou estat
`execState coinTrigger Locked` 👉 **Unlocked**

22 / 29

Contingut

- Mònade llista
- Mònade estat
- Combinació de mònades

23 / 29

Petit LP I

Tipus

```
data Expr = Val Int
| Var String
| Add Expr Expr
| Sub Expr Expr
| Mul Expr Expr
| Div Expr Expr
deriving (Show)

data Accio = Ass String Expr
| View Expr
deriving (Show)
```

Exemple

```
View (Val 2)           ➡ Just 2
Ass "a" (Val 3)        ➡ Just 3
View (Add (Var "a") (Val 2)) ➡ Just 5
👉 Estat final: [("a",Just 3)]
```

24 / 29

Petit LP II

Requeriments

```
import Control.Monad
import Control.Monad.State
```

Estat: taula de símbols

- tipus:

```
type TS = [(String, Maybe Int)]
```

- nou símbol:

```
addEntry :: Eq a => a -> b -> [(a, b)] -> [(a, b)]
addEntry k v l = (k, v) : filter (\p -> fst p /= k) l
```

- consulta símbol:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

25 / 29

Petit LP III

Expressions

```
eval :: Expr -> State TS (Maybe Int)
eval (Val x) = return (Just x)
eval (Var x) = do
    ts <- get
    return (join $ lookup x ts)           -- recuperem la taula de símbols
                                                -- join $ Just $ Just 2 ➔ Just 2
eval (Add x y) = evalf (+) x y
eval (Sub x y) = evalf (-) x y
eval (Mul x y) = evalf (*) x y
eval (Div x y) = do
    ts <- get
    let vlx = evalState (eval x) ts      -- recuperem la taula de símbols
    let vly = evalState (eval y) ts      -- només volem el valor, l'estat no canvia
    return $ evalDiv vlx vly             -- idem
```

26 / 29

Petit LP IV

Funcions auxiliars d'`eval`

```
evalf :: (Int -> Int -> Int) -> Expr -> Expr -> State TS (Maybe Int)
evalf f x y = do
    ts <- get                      -- recuperem la taula de símbols
    let vlx = evalState (eval x) ts   -- només volem el valor, l'estat no canvia
    let vly = evalState (eval y) ts   -- idem
    return $ liftM2 f vlx vly         -- vlx i vly són Maybe Int

evalDiv :: Maybe Int -> Maybe Int -> Maybe Int
evalDiv x y = do
    x' <- x
    y' <- y
    if y' == 0
    then Nothing
    else return $ div x' y'
```

27 / 29

Petit LP V

Accions

```
exec :: Accio -> State TS (Maybe Int)
exec (Ass s x) = do
    ts <- get
    let (vlx, tsx) = runState (eval x) ts
    put $ addEntry s vlx tsx          -- actualització de la taula de símbols
    return vlx
exec (View x) = do
    ts <- get
    let vl = evalState (eval x) ts
    return vl
```

Ús

```
runState (exec (View (Val 2))) []
👉 (Just 2, [])
```

28 / 29

Petit LP VI

Enllaç

```
let l = [Ass "a" (Val 3), View (Add (Var "a") (Val 2))]

runState (exec (l!!0) >>= \x -> exec (l!!1) >>= \y -> return [x, y]) []
👉 ([Just 3,Just 5],[("a",Just 3)])      -- (valors resultants, TS final)

runState (mapM exec l) []
👉 ([Just 3,Just 5],[("a",Just 3)])
```

Un altre exemple

```
let l = [View (Val 2), Ass "a" (Val 3), View (Add (Var "a") (Val 2))]
runState (mapM exec l) []
👉 ([Just 2,Just 3,Just 5],[("a",Just 3)])
```

Llenguatges de Programació

Raonament equacional



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

2 / 31

Raonament equacional

El raonament equacional permet reflexionar sobre programes funcionals per tal d'establir propietats usant igualtats i substitucions matemàtiques.

Sovint s'estableixen equivalències entre funcions.

Aquestes es poden aprofitar per:

- millorar l'eficiència de programes.
- verificar programes:
demonstrar que un programa és correcte respecte la seva especificació.
- derivar programes:
deduir el programa formalment a partir de l'especificació.

3 / 31

Exemple: Multiplicació de complexos

$(a+bi)*(c+di)$ es pot calcular amb 4 productes de reals:

```
def mult(a, b, c, d):
    re1 = a*c - b*d
    im1 = b*c + a*d
    return (re1, im1)
```

Gauss va descobrir que només calien 3 productes:

```
def gauss(a, b, c, d):
    t1 = a * c
    t2 = b * d
    re2 = t1 - t2
    im2 = (a + b) * (c + d) - t1 - t2
    return (re2, im2)
```

Podem comprovar matemàticament l'equivalència entre les dues funcions:

$$\begin{aligned} (\text{re2}, \text{im2}) &= (t1 - t2, (a + b) * (c + d) - t1 - t2) \\ &= (a*c - b*d, (a + b) * (c + d) - a*c + b*d) \\ &= (\text{re1}, a*c + a*d + b*c + b*d - a*c - b*d) \\ &= (\text{re1}, b*c + a*d) \\ &= (\text{re1}, \text{im1}) \end{aligned}$$

4 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

5 / 31

Associativitat de la composició

Propietat: $f . (g . h) = (f . g) . h$.

Definició:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$(f_1 . f_2) x = f_1 (f_2 x)$$



Demostració: Sigui qualsevol dada x , llavors:

$$\begin{aligned} (f . (g . h)) x &= \\ &\quad \text{-- definició de } . \\ &= f ((g . h) x) \\ &\quad \text{-- definició de } . \\ &= f (g (h x)) \\ &\quad \text{-- definició de } . \\ &= (f . g) (h x) \\ &\quad \text{-- definició de } . \\ &= ((f . g) . h) x \end{aligned}$$

$$\Rightarrow f . (g . h) = (f . g) . h$$

Observació: S'ha usat en els dos sentits.

6 / 31

Involució de la negació

Propietat: `not . not = id.`

```
not :: Bool -> Bool  
not True = False  
not False = True
```

1
2

```
id :: a -> a  
id x = x
```

3

Demostració: Hi ha dos casos:

```
(not . not) True =  
    -- definició de .  
= not (not True)  
    -- 1  
= not False  
    -- 2  
= True  
    -- 3  
= id True
```

7 / 31

Involució de la negació

```
(not . not) False =  
    -- definició de .  
= not (not False)  
    -- 2  
= not True  
    -- 1  
= False  
    -- 3  
= id False
```

$\Rightarrow \text{not . not} = \text{id}$

Observació: S'han cobert tots els possibles casos explícitament.

8 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

9 / 31

map és distributiva sobre ++

Propietat: $\text{map } f \ (xs \ ++ \ ys) = \text{map } f \ xs \ ++ \ \text{map } f \ ys$.

$$\begin{aligned} \text{map } f \ [] &= [] & 1 \\ \text{map } f \ (x:xs) &= f \ x : \text{map } f \ xs & 2 \end{aligned}$$

$$\begin{aligned} [] \ ++ \ ys &= ys & 3 \\ (x:xs) \ ++ \ ys &= x : xs \ ++ \ ys & 4 \end{aligned}$$

Demostració: Inducció sobre xs .

A Cas base: $xs = []$.

$$\begin{aligned} \text{map } f \ ([] \ ++ \ ys) &= \\ &\quad \text{-- 3} \\ &= \text{map } f \ ys \\ &\quad \text{-- 3} \\ &= [] \ ++ \ \text{map } f \ ys \\ &\quad \text{-- 1} \\ &= \text{map } f \ [] \ ++ \ \text{map } f \ ys \end{aligned}$$

map és distributiva sobre ++

Demostració: Inducció sobre xs.

B Cas inductiu: $xs = z:zs$. HI: $\text{map } f (zs ++ ys) = \text{map } f zs ++ \text{map } f ys$.

```
map f (xs ++ ys) =
    -- definició de xs
    = map f ((z:zs) ++ ys)
    -- 4
    = map f (z : zs ++ ys)
    -- 2
    = f z : map f (zs ++ ys)
```

```
map f xs ++ map f ys =
    -- definició de xs
    = map f (z:zs) ++ map f ys
    -- 2
    = (f z : map f zs) ++ map f ys
    -- 4
    = f z : (map f zs ++ map f ys)
    -- hipòtesi d'inducció
    = f z : map f (zs ++ ys)
```

$$\Rightarrow \text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys$$

11 / 31

map és distributiva sobre ++

Demostració:

S'ha demostrat per inducció sobre xs:

A Cas base: $xs = []$.

B Cas inductiu: $xs = z:zs$.

Per tant,

$$\text{map } f (xs ++ ys) = \text{map } f xs ++ \text{map } f ys$$

12 / 31

Involució del revessat

Propietat: `reverse . reverse = id.`

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

Lema 1:

```
reverse [x] = [x]
```

Demostració: exercici (fàcil!).

Lema 2:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Demostració: exercici (inducció sobre `xs`).

13 / 31

Involució del revessat

Propietat: `reverse . reverse = id.`

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

Demostració:

A Cas base: `xs = []`.

```
(reverse . reverse) [] =
  -- definició de .
  = reverse (reverse [])
    -- definició de reverse
  = reverse []
    -- definició de reverse
  = []
    -- definició de id
  = id []
```

14 / 31

Involució del revessat

B Cas inductiu: $\text{xs} = \text{z}: \text{zs}$.

Hipòtesi d'inducció: (`reverse . reverse`) $\text{zs} = \text{id}$ zs

```
(reverse . reverse) xs =
  -- definició de xs
  = (reverse . reverse) (z:zs)
    -- definició de .
    = reverse (reverse (z:zs))
      -- definició de reverse
    = reverse (reverse zs ++ [z])
      -- lema 2
    = reverse [z] ++ reverse (reverse zs)
      -- lema 1
  = [z] ++ reverse (reverse zs)
    -- definició de .
  = [z] ++ (reverse . reverse) zs
    -- hipòtesi d'inducció i definició de id
  = [z] ++ zs
    -- definició de ++
  = z:zs
    -- definició de xs
= xs
```

15 / 31

Involució del revessat

Per tant, queda demostrat que:

`reverse . reverse = id`

tal com es volia.

16 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

17 / 31

map per arbres

```
data Arbin a = Empty | Node a (Arbin a) (Arbin a)

treeMap :: (a -> b) -> (Arbin a) -> (Arbin b)

treeMap _ Empty = Empty
treeMap f (Node x l r) = Node (f x) (treeMap f l) (treeMap f r)
```

Propietat: `treeMap id = id`.

Demostració: Inducció sobre l'arbre t:

A Cas base: `t = Empty`.

```
treeMap id Empty =
    -- definició de treeMap
= Empty
    -- definició de id
= id Empty
```

18 / 31

map per arbres

Demostració: Inducció sobre l'arbre t:

B Cas inductiu: $t = \text{Node } x \text{ } l \text{ } r$.

HI: $\text{treeMap id } l = l$ i $\text{treeMap id } r = r$

```
treeMap id t =
    -- definició de t
    = treeMap id (Node x l r)
    -- definició de treeMap
    = Node (id x) (treeMap id l) (treeMap id r)
    -- definició de id
    = Node x (treeMap id l) (treeMap id r)
    -- HI
    = Node x l r
    -- definició de t
    = t
```

Per tant,

$\text{treeMap id} = \text{id}$.

19 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

20 / 31

Millorant reverse

Especificació:

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

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

1

2

Problema: Donat que `++` necessita temps $O(n)$, `reverse` necessita temps $O(n^2)$ 😞.

Podem revessar més ràpidament? 🤔

21 / 31

Millorant reverse

Considerem una *generalització* de `reverse`:

```
revcat :: [a] -> [a] -> [a]
```

```
revcat xs ys = reverse xs ++ ys
```

3

Llavors:

```
reverse xs = revcat xs []
-- per 3 i per definició de ++
```

Podem ara definir `revcat`? 🤔

22 / 31

Millorant reverse

A Cas base: `xs = []`.

```
revcat [] ys =
  -- 3 i definició de ++
= reverse [] ++ ys
  -- 1
= [] ++ ys
  -- definició de ++
= ys
```

B Cas inductiu: `xs = z:zs`.

```
revcat (z:zs) ys =
  -- 3
= reverse (z:zs) ++ ys
  -- 2
= (reverse zs ++ [z]) ++ ys
  -- associativitat de ++
= reverse zs ++ ([z] ++ ys)
  -- 3
= revcat zs ([z] ++ ys)
  -- lema sobre ++
= revcat zs (z:ys)
```

23 / 31

Millorant reverse

Per tant,

```
reverse xs = revcat xs []
where
  revcat [] ys = ys
  revcat (x:xs) ys = revcat xs (x:ys)
```

funciona en temps lineal! 😊

A més, `revcat` és recursiva final, per tant necessita espai constant.

- Una funció recursiva és *final* si la crida recursiva és el darrer pas que fa
 - es pot canviar el `call` per un `jmp`.

24 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- **Sumari**
- Exercicis

25 / 31

Sumari

La programació funcional permet raonar senzillament sobre els programes usant equacions i mètodes matemàtics.

Trobar equivalències entre expressions:

- Pot ser útil per demostrar propietats i correctesa.
- Pot ser útil per donar lloc a optimitzacions.

Demostrar equivalències vol pràctica (igual que la programació).

Per fer-ho amb èxit sovint cal:

- ser exhaustiu.
- usar definicions en els dos sentits.
- utilitzar inducció.
- aprofitar les definicions recursives.
- demostrar resultats auxiliars.

26 / 31

Contingut

- Introducció
- Base
- Inducció
- Inducció amb arbres
- Millora eficiència
- Sumari
- Exercicis

27 / 31

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Demostreu que `xs ++ [] = xs = [] ++ xs.`
2. Demostreu que `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.`
3. Demostreu que `reverse (xs ++ ys) = reverse ys ++ reverse xs.`
4. Demostreu que `length (xs ++ ys) = length xs + length ys..`
5. Demostreu que `take n xs ++ drop n xs = xs.`
6. Demostreu que `drop n (drop m xs) = drop (m + n) xs.`
7. Demostreu que `head . map f = f . head.` Aneu amb compte amb el cas de la llista buida. Quin interès té aquesta equivalència?
8. Demostreu que `filter p (xs ++ ys) = filter p xs ++ filter p ys.`
9. Demostreu que `foldl f e xs = foldr (flip f) e (reverse xs).`
10. Demostreu que `foldl (@) e xs = foldr (<>) e xs` quan $(x \leftrightarrow y) @ z = x \leftrightarrow (y @ z)$ i $e @ x = x \leftrightarrow e.$

28 / 31

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Considereu aquest tipus pels naturals:

```
data Nat = Z | S Nat
```

Definiu les funcions següents amb significat obvi:

```
intToNat :: Int -> Nat
natToInt :: Nat -> Int
add      :: Nat -> Nat -> Nat      -- no es pot usar la suma d'Ints!.
```

- Demostreu que `intToNat` i `natToInt` són inverses l'una de l'altra.
- Demostreu que `Z` és element neutre (per la dreta i per l'esquerra) de `add`.
- Demostreu que `add (S x) y = add x (S y)`.
- Demostreu l'associativitat de `add`.
- Demostreu la commutativitat de `add`.

29 / 31

Exercicis

(Assumiu que totes les EDs són finites i els tipus correctes)

1. Definiu arbres binaris amb una operació `size` i una operació `mirror`.
Demostreu que `size . mirror = size`.

30 / 31

Exercicis

Una llista es diu que és *supercreixent* si cada element és més gran que la suma dels seus anteriors:

```
superCreixent :: Num a, Ord a => [a] -> Bool  
superCreixent [] = True  
superCreixent (x:xs) = superCreixent && x > sum xs
```

1. Mostreu que `superCreixent` funciona en temps quadràtic.
2. Definiu una funció `superCreixent' :: [a] -> (Bool, a)` que, donada una llista, retorni si és *supercreixent* i quina és la seva suma.
3. Escriviu `superCreixent'` en termes de `superCreixent`.
4. Escriviu `superCreixent` en termes de `superCreixent'`.
5. Deriveu `superCreixent'`.
6. Doneu el temps d'execució de `superCreixent'`.

Introducció a la compilació

```
class ConsultantAvailability {
  constructor() {
    this.secondHeader = {};
  }

  getSecondHeader(item) {
    let data = {
      label: moment(item, 'ddd').format('ddd'),
      dateFrom: item.dateFrom.clone().add(1, 'days').format(this.dateFormat)
    };
    this.firstHeader.push(data);
  }

  getSecondHeaderByWeek() {
    let data = [];
    this.firstHeader.forEach(item => {
      let secondHeaderByWeek = {};
      secondHeaderByWeek.push({
        firstHeader: item,
        secondHeader: item
      });
    });
    this.secondHeader[0] = secondHeaderByWeek;
  }

  map(consultants, consultant) {
    let data = [];
    consultants.forEach(item => {
      let dataItem = moment(item.data.date + ' ' + item.timeSlot, 'DD/MM/YYYY HH:mm').format('YYYY-MM-DDTHH:mm');
      consultant[consultant.data.label + ' ' + dataItem] = item.consultantIsAvailable;
    });
    return consultants;
  }
}

const consultantAvailability = new ConsultantAvailability();
const consultantsAvailables = document.getElementById('consultants-availables');
```

```
<template>
  <div>
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Available</th>
        </tr>
      </thead>
      <tbody>
        <tr v-for="consultant in consultantsAvailables">
          <td>{{consultant.name}}</td>
          <td>{{consultant.available}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 87

Contingut

- Introducció
- Visió general
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

Objectius

- Conèixer l'estructural general d'un compilador, les seves principals etapes, i la seva organització.
- Conèixer l'existència d'eines per ajudar a crear compiladors (usarem [ANTLR](#)).
- A la pràctica, ens limitem a crear petits processadors de llenguatges:
 1. Definició del vocabulari,
 2. Definició de la gramàtica,
 3. Generació de l'arbre de sintaxi abstracta,
 4. Interpretació a través del recorregut de l'arbre.
- El curs de Compiladors aprofundeix molts més els continguts.
- El curs de Teoria de la Computació n'ofereix els fonaments teòrics.

3 / 87

Crèdits

Gran part del material d'aquestes diapositives sobre compilació s'ha extret de les que va elaborar el professor Stephen A. Edwards (Universitat de Columbia) per l'assignatura COMS W4115 (Programming Languages and Translators) i que el professor Jordi Cortadella (UPC) va adaptar per l'assignatura de Compiladors. També s'ha extret material de les transparències del professor Fernando Orejas (UPC).

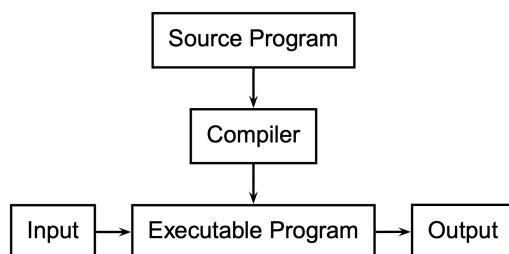
4 / 87

Contingut

- Introducció
- Visió general
 - Processadors de llenguatges
 - Sintaxi
 - Semàntica
 - Flux de compilació
 - Eines
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

5 / 87

Compiladors



Un compilador és un programa que tradueix programes escrits en un LP d'alt nivell a codi màquina (o, en general, a codi de baix nivell).

Exemples: GCC, CLANG, go, ghc, ...

6 / 87

Compiladors

Compilació en C

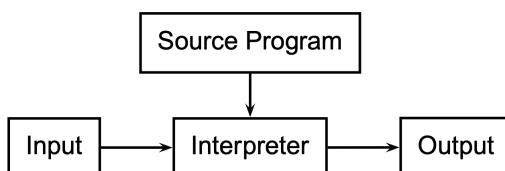
```
int func(int a, int b) {
    a = a + b;
    return a;
}
```

```
gcc -S prova.c
```

```
_func:
    pushq  %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    -8(%rbp), %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    popq    %rbp
    retq
```

7 / 87

Intèrprets



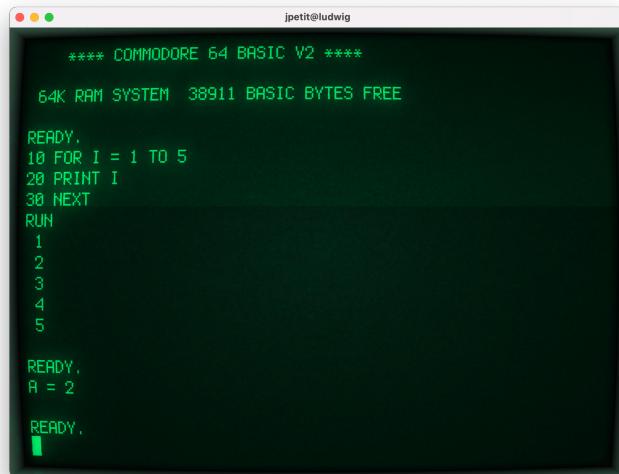
Un intèrpret és un programa que executa directament instruccions escrites en un LP.

Exemples: Python, PHP, Perl, ghci, BASIC, Logo...

8 / 87

Intèrprets

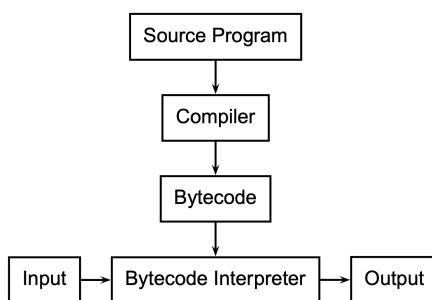
Sessió amb l'intèrpret de BASIC al Commodore 64 (emulat al Mac ⚡!)



The screenshot shows a terminal window titled "jpettit@ludwig" running the Commodore 64 BASIC V2 interpreter. The screen displays the following text:
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
10 FOR I = 1 TO 5
20 PRINT I
30 NEXT
RUN
1
2
3
4
5
READY.
A = 2
READY.

9 / 87

Intèrprets de bytecode



Variant entre els compiladors i els intèrprets.

- El bytecode és un codi intermedi més abstracte que el codi màquina.
- Augmenta la portabilitat i seguretat i facilita la interpretació.
- Una màquina virtual interpreta programes en bytecode.

Exemples: Java, Python, ...

Intèrprets de bytecode

CPython per a Python

```
>>> import dis # desensamblador
>>> dis.dis("a = a + b")
  1  0 LOAD_NAME      0 (a)
  2  0 LOAD_NAME      1 (b)
  4  0 BINARY_ADD
  6  0 STORE_NAME     0 (a)
  8  0 LOAD_CONST     0 (None)
 10 0 RETURN_VALUE
```

La VM de CPython usa tres tipus de piles:

- Call stack: Guarda l'estructura principal de l'execució d'un programa en Python.

Hi ha *frame* per a cada crida oberta a una funció. Cada crida a una funció empila un nou *frame* i cada sortida de funció el desempila. És on es desen les variables locals.

- A cada *frame*, hi ha un evaluation stack: És on es fa l'avaluació de les expressions, ficant paràmetres i extraient resultats.
- A cada *frame*, també hi ha un block stack: És on es realitza l'execució de les instruccions (condicionals, bucles, try/excepts, withs, continues, breaks, ...)

11 / 87

Intèrprets de bytecode

JVM per a Java

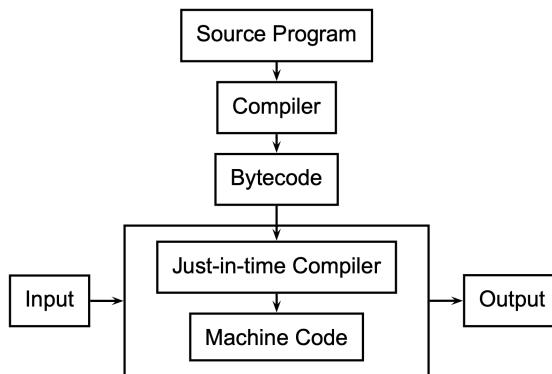
```
public static void func(int a, int b) {
    a = a + b;
}
```

```
javap -v prova.class
```

```
public static void func(int, int);
descriptor: (II)V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
  stack=2, locals=2, args_size=2
    0: iload_0
    1: iload_1
    2: iadd
    3: istore_0
    4: return
```

12 / 87

Compiladors *just-in-time*



La compilació just-in-time (JIT) compila fragments del programa durant la seva execució.

Un analitzador inspecciona el codi executat per veure quan val la pena compilar-lo.

Exemples: Julia, V8 per Javascript, JVM per Java, ...

13 / 87

Compiladors *just-in-time*

Numba tradueix funcions en Python a codi màquina i optimitzat usant LLVM en temps d'execució.

```
import numba
import random

@numba.jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

14 / 87

Preprocessadors

Un preprocessador prepara el codi font d'un programa abans que el compilador el vegi.

- Expansió de macros
- Inclusió de fitxers
- Compilació condicional
- Extensions de llenguatge

Exemples: cpp, m4, ...

15 / 87

Preprocessadors

El preprocessador de C

```
#include <stdio.h>
#define min(x, y) ((x)<(y))?(x):(y)
#ifndef DEFINE_BAZ
int baz();
#endif
void foo() {
    int a = 1;
    int b = 2;
    int c;
    c = min(a,b);
}
```

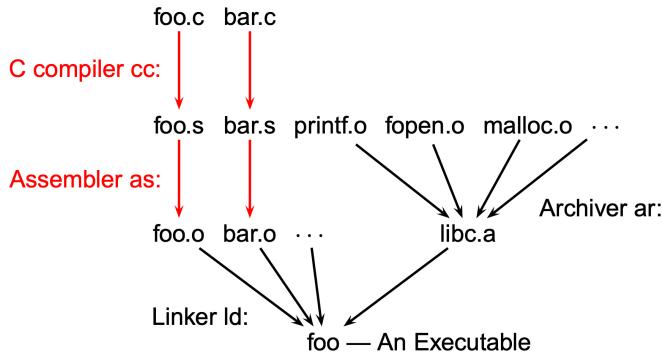
```
gcc -E programa.c
```

```
extern int printf(char*, ...);
/* moltes més línies de stdio.h
void foo() {
    int a = 1;
    int b = 2;
    int c;
    c = ((a)<(b))?(a):(b);
}
```

16 / 87

Ecosistema

Els processadors de llenguatges viuen en un ecosistema gran i complex: preprocessadors, compiladors, enllaçadors, gestors de llibreries, ABIs (application binary interfaces), formats d'executables, ...



17 / 87

Contingut

- Introducció
- Visió general
 - Processadors de llenguatges
 - Sintaxi
 - Semàntica
 - Flux de compilació
 - Eines
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

18 / 87

Sintaxi

La sintaxi d'un llenguatge de programació és el conjunt de regles que defineixen les combinacions de símbols que es consideren construccions correctament estructurades.

Jove xef, porti whisky amb quinze glaçons d'hidrogen, coi!

➡ frase sintàcticament correcta en català, però no és un programa en Java.

```
class Foo {  
    public int j;  
    public int foo(int k) {  
        return j + k;  
    }  
}
```

➡ programa sintàcticament correcte en Java, però no és un programa en C.

19 / 87

Sintaxi

Sovint s'especifica la sintaxi utilitzant una gramàtica lliure de context (*context-free grammar*).

Els elements més bàsics ("paraules") s'especifiquen a través d'expressions regulars.

Exemple per expressions algebraiques:

```
expr → NUM  
      | '(' expr ')'  
      | expr '+' expr  
      | expr '-' expr  
      | expr '*' expr  
      | expr '/' expr  
  
NUM → [0-9]+ ( '.' [0-9]+ )
```

20 / 87

Contingut

- Introducció
- Visió general
 - Processadors de llenguatges
 - Sintaxi
 - Semàntica
 - Flux de compilació
 - Eines
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

21 / 87

Semàntica

La semàntica d'un LP descriu què significa un programa ben construït.

```
def fib(n):  
    a, b = 0, 1  
    for i in range(n):  
        a, b = b, a + b  
    return a
```

➡ La semàntica d'aquesta funció en Python és el càcul de l' n -èsim nombre de Fibonacci.

22 / 87

Semàntica

A vegades, les construccions sintàcticament correctes poden ser semànticament incorrectes.

L'arc de Sant Martí va saltar sobre el planeta pelut.

- ➡ sintàcticament correcta en català, però sense sentit.

```
class Foo {  
    int bar(int x) { return Foo; }  
}
```

- ➡ sintàcticament correcte en Java, però sense sentit.

23 / 87

Semàntica

A vegades, les construccions sintàcticament correctes poden ser ambigües.

Han posat un banc a la plaça.

- ➡ sintàcticament correcta en català, però ambigua

```
class Bar {  
    public float foo() { return 0; }  
    public int foo() { return 0; }  
}
```

- ➡ sintàcticament correcte en Java, però ambigu.

24 / 87

Semàntica

Hi ha bàsicament dues maneres d'especificar formalment la semàntica:

- Semàntica operacional: defineix una màquina virtual i com l'execució del programa canvia l'estat de la màquina.
- Semàntica denotacional: mostra com construir una funció que representa el comportament del programa (és a dir, una transformació d'entrades a sortides) a partir de les construccions del LP.

La majoria de definicions de semàntica per a LPs utilitzen una semàntica operacional descrita informalment en llenguatge natural.



25 / 87

Contingut

- Introducció
- Visió general
 - Processadors de llenguatges
 - Sintaxi
 - Semàntica
 - Flux de compilació
 - Eines
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

26 / 87

Etapes

- Front end
 - preprocessador
 - analitzador lèxic (escàner)
 - analitzador sintàctic (parser)
 - analitzador semàntic
- Middle end
 - analitzador de codi intermedi
 - optimitzador de codi intermedi
- Back end
 - generador de codi específic
 - optimitzador de codi específic

27 / 87

Entrada

```
int gcd(int a, int b) {  
    while (a != b) {  
        if (a > b) a -= b; else b -= a;  
    }  
    return a;  
}
```

El compilador veu una seqüència de caràcters:

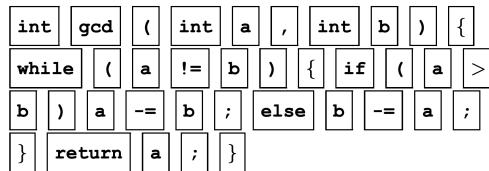
```
i n t u g c d ( i n t u a , u i n t u b ) u { u u w h i l e u  
( a u ! = u b ) u { u u u u u i f u ( a u > u b ) u a u - = u b  
; u e l s e u b u - = u a ; u u } u u r e t u r n u a ; u } u
```

28 / 87

Analitzador lèxic

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

L'analitzador lèxic (escàner) agrupa els caràcters en "paraules" (tokens) i elimina blancs i comentaris.

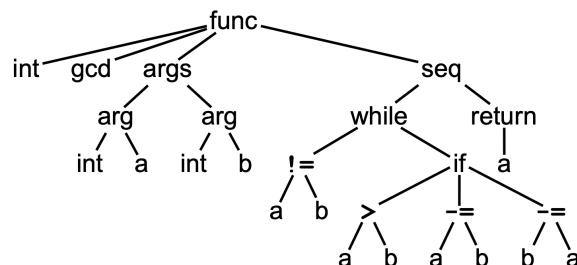


29 / 87

Analitzador sintàctic

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

L'analitzador sintàctic (parser) construeix un arbre de sintaxi abstracta (AST) a partir de la seqüència de tokens i les regles sintàctiques.



Les paraules clau, els separadors, parèntesis i blocs s'eliminen.

30 / 87

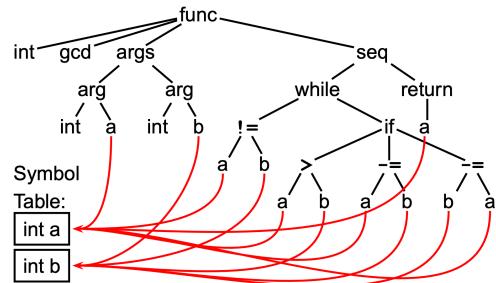
Analitzador semàntic

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

L'analitzador semàntic recorre l'AST i:

- crea la taula de símbols,
- assigna memòria a les variables,
- comprova errors de tipus,
- resol ambigüïtats.

El resultat és la taula de símbols i un AST decorat.



31 / 87

Generador de codi intermedi

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

El generador de codi tradueix el programa a codi de tres adreces (ensamblador idealitzat amb infinitat de registres).

```
L0:    sne $1, a, b          # signed not equal
       seq $0, $1, 0          # signed equal
       btrue $0, L1            # while a != b
       sl $3, b, a             # signed less
       seq $2, $3, 0
       btrue $2, L4            #     if a < b
       sub a, a, b              #         a -= b
       jmp L5
L4:    sub b, b, a              #         b -= a
L5:    jmp L0
L1:    ret a                  # return a
```

32 / 87

Optimitzador de codi intermedi

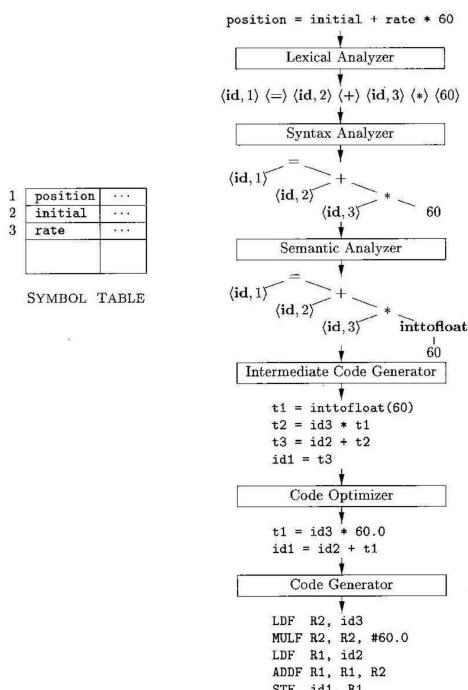
```
L0:    sne $1, a, b          # signed not equal
       seq $0, $1, 0          # signed equal
       btrue $0, L1           # while a != b
       sl $3, b, a           # signed less
       seq $2, $3, 0
       btrue $2, L4           # if a < b
       sub a, a, b           # a -= b
       jmp L5
L4:    sub b, b, a           # b -= a
L5:    jmp L0
L1:    ret a                # return a
```

El back end tradueix i optimitza el codi de tres adreces a l'arquitectura desitjada:

```
gcd:   pushl %ebp          # Save FP
       movl %esp,%ebp
       movl 8(%ebp),%eax
       movl 12(%ebp),%edx
.L8:   cmpl %edx,%eax
       je .L3               # while a != b
       jle .L5               # if (a < b)
       subl %edx,%eax
       jmp .L8
.L5:   subl %eax,%edx
       jmp .L8
.L3:   leave                # Restore SP, BP
       ret                  # return a
```

33 / 87

Sumari



34 / 87

Contingut

- Introducció
- Visió general
 - Processadors de llenguatges
 - Sintaxi
 - Semàntica
 - Flux de compilació
 - Eines
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

35 / 87

Eines

Per construir un compilador no es parteix de zero.

Hi ha moltes eines que donen suport.

Exemples:

- ANTLR, donades les especificacions lèxiques i sintàctiques del LP, construeix automàticament l'escàner, l'analitzador i l'AST.
- LLVM ofereix una col·lecció d'eines modulars reutilitzables pels backends dels compiladors.

36 / 87

Contingut

- Introducció
- Visió general
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

37 / 87

Anàlisi lèxica

L'analitzador lèxic (o escàner) converteix una seqüència de caràcters en una seqüència de tokens:

- identificadors,
- literals (nombres, textos, caràcters)
- paraules clau,
- operadors,
- puntuació...

f o o _ = _ a + _ bar(2, _ q);						
ID	EQUALS	ID	PLUS	ID	LPAREN	NUM
COMMA	ID	LPAREN	SEMI			

Token	Lexemes	Pattern
EQUALS	=	an equals sign
PLUS	+	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

38 / 87

Objectius

- Simplificar la feina de l'analitzador sintàctic.

El parser no té en compte els noms dels identificadors, només li preocuten els tokens (*supercalifragilisticexpialidocious* → ID).

- Descartar detalls irrelevants: blancs, comentaris, ...
- Els escàners són molt més ràpids que els parsers.

39 / 87

Descripció de tokens

Per especificar els tokens s'utilitzen conceptes de la teoria de llenguatges:

Alfabet: un conjunt finit de símbols.

Exemples: {0, 1}, {A, B, ..., Z}, ASCII, Unicode, ...

Paraula: una seqüència finita de símbols de l'alfabet.

Exemples: ϵ (la paraula buida), *foo*, $\alpha\beta\gamma$.

Llenguatge: Un conjunt de paraules sobre un alfabet.

Exemples: \emptyset (el llenguatge buit), { 1, 11, 111, 1111 }, tots els mots anglesos, tots els identificadors (textos que comencen amb una lletra seguida per lletres o díigits).

40 / 87

Operacions sobre llenguatges

Exemple: $L = \{ \epsilon, wo \}$, $M = \{ man, men \}$

Concatenació: Una paraules d'un llenguatge seguida d'una paraula de l'altre llenguatge.

$$L M = \{ man, men, woman, women \}$$

Unió: Totes les paraules de cada llenguatge.

$$L \cup M = \{ \epsilon, wo, man, men \}$$

Clausura de Kleene: Zero o més concatenacions.

$$M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \cup \dots = \{ \epsilon, man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, \dots \}$$

41 / 87

Expressions regulars

Les expressions regulars descriuen llenguatges a partir de tokens sobre un alfabet Σ .

1. ϵ és una expressió regular que denota $\{\epsilon\}$.
2. Si $a \in \Sigma$, a és una expressió regular que denota $\{a\}$.
1. Si r i s denoten els llenguatges $L(r)$ i $L(s)$,
 - $(r) | (s)$ denota $L(r) \cup L(s)$
 - $(r)(s)$ denota $\{tu : t \in L(r), u \in L(s)\}$
 - (r^*) denota la clausura transitiva de $L(r)$

42 / 87

Expressions regulars

Exemples:

$$\Sigma = \{a, b\}$$

RE	Language
$a b$	$\{a, b\}$
$(a b)(a b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \dots\}$
$(a b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\}$
$a a^*b$	$\{a, b, ab, aab, aaab, aaaab, \dots\}$

43 / 87

Generadors d'escàners

Les expressions regulars s'usen en eines per crear compiladors:

- lex, ANTLR, ...

I també,

- en comandes del SO per tractar fitxers (`grep`, `sed`, ...)
- en llibreries d'LPs per tractar textos (`re` en Python, directament en Javascript, ...)

44 / 87

Generadors d'escàners

A partir de la definició lèxica, l'escàner és un autòmat determinista que produeix com a sortida els tokens reconeguts.

Construcció:

Regles amb expressions regulars



Autòmats finits no deterministes



Autòmat finit determinista



Taules

45 / 87

Analitzador lèxic d'ANTLR

```
RET      : 'return' ;  
LPAREN  : '(' ;  
RPAREN  : ')' ;  
ADD     : '+' ;  
SUB     : '-' ;  
MUL     : '*' ;  
DIV     : ';' ;  
  
DIGIT   : '0'..'9' ;  
LETTER  : [a-zA-Z] ;  
  
NUMBER  : (DIGIT)+ ;  
IDENT   : LETTER (LETTER | DIGIT)* ;  
  
WS      : [ \t\n]+ -> skip ;
```

Referència

- els noms dels tokens han de ser en majúscules
- textos entre cometes representen aquell text
- + indica concatenació
- | indicaunió
- * indica zero o més
- + indica un o més
- ? indica un o cap
- .. indica rangs
- es poden agrupar elements amb parèntesis
- l'skip no reporta el token

46 / 87

grep

La comanda **grep** (*global regular expression print*) permet cercar patrons en texts.

```
grep 'jpetit' /etc/passwd
```

Imatge: Dave Child, cheatography.com

Anchors	Assertions	Groups and Ranges
* Start of string, or start of line in multi-line pattern	?= Lookahead assertion	Any character except new line (\n)
\A Start of string	?< Negative lookahead	(a b) a or b
\$ End of string, or end of line in multi-line pattern	?> Lookbehind assertion	(...) Group
\Z End of string	?>= Negative lookbehind	(?:-) Passive (non-capturing) group
\b Word boundary	?>> Once-only Subexpression	[abc] Range (a or b or c)
\B Not word boundary	?{n} Condition [if then]	[^abc] Not (a or b or c)
\c Start of word	?{n} Condition [if then else]	[a-q] Lower case letter from a to q
\d End of word	?# Comment	[A-Q] Upper case letter from A to Q
Character Classes	Quantifiers	Digit from 0 to 7
\c Control character	* 0 or more {3} Exactly 3	[0-7] Digit from 0 to 7
\s White space	+ 1 or more {3,} 3 or more	x Group/subpattern number "x"
\S Not white space	?{0,1} {3,5} 3, 4 or 5	Ranges are inclusive.
\d Digit	Add a ? to a quantifier to make it ungreedy.	
\D Not digit		
\w Word		
\W Not word		
\x Hexadecimal digit		
\o Octal digit		
POSIX	Escape Sequences	Pattern Modifiers
[upper]	\ Escape following character	g Global match
[lower]	<Q> Begin literal sequence	i Case-insensitive
[alpha]	<E> End literal sequence	m+ Multiple lines
[alnum]	"Escaping" is a way of treating characters which have a special meaning in regular expressions literally, rather than as special characters.	s+ Treat string as single line
[digit]		x+ Allow comments and whitespace in pattern
[xdigit]		e+ Evaluate replacement
[punct]		U+ Ungreedy pattern
[blank]		PCRE modifier
[space]		
[control]		
[graph]		
[print]		
[word]		
Common Metacharacters	Special Characters	String Replacement
^ [- . \$	\n New line	\\$n nth non-passive group
[* () \ \	\r Carriage return	\\$2 "xyz" in "(abc)xyz\\$"
+)] ?	\t Tab	\\$1 "xyz" in "?abc)xyz\\$"
< >	\v Vertical tab	\\$ Before matched string
	\f Form feed	\\$` After matched string
	\xxx Octal character xxx	\\$+ Last matched string
	\xhh Hex character hh	\\$& Entire matched string
		Some regex implementations use \ instead of \\$.

47 / 87

re

El mòdul **re** de Python proporciona operacions sobre expressions regulars.

```
>>> import re

>>> p = re.compile('[a-z]+')
>>> p.match("7654386732(.)")
None
>>> m = p.match('unicorn')
<re.Match object; span=(0, 5), match='unicorn'>
>>> m.group()
'unicorn'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)

>>> re.search('casa', 'vaig a casa a dormir')
<re.Match object; span=(7, 11), match='casa'>

>>> re.sub('[0-9]+', 'molt', 'val 350 euros')
'val molt euros'
```

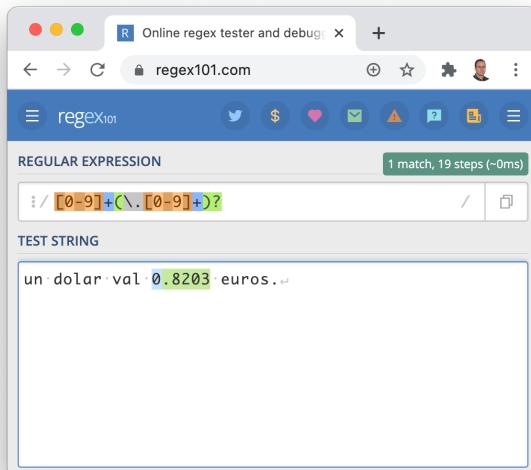
Tutorial

48 / 87

Testing d'expressions regulars

Some people, when confronted with a problem, think "*I'll use regular expressions*". Now they have two problems.

— Jamie Zawinski, 1997



49 / 87

Exercicis

Definiu expressions regulars pels llenguatges següents:

1. Identificadors de C++: poden contenir lletres, díigits i subratllats però no poden començar per un dígit.
2. Nombres en coma flotant. Si s'utilitza un punt decimal, un dígit decimal és obligatori.

Exemples: 3.1416, -3e4, +1.0e-5, .567e+8, ...

3. Totes les paraules amb alfabet {a, b, c} en les quals la primera aparició de b sempre és precedida per, com a mínim, una aparició de a.
4. Totes les paraules amb minúscules que contenen les cinc vocals en ordre (cada vocal només pot aparèixer un sol cop).

Example: zfaehipojksuj

5. Totes les paraules amb minúscules en les quals les lletres es troben en ordre lexicogràfic ascendent.

Examples: afhmnnqsyz, abcdz, dgky, ... Contraexemple: bdeaz.

50 / 87

Contingut

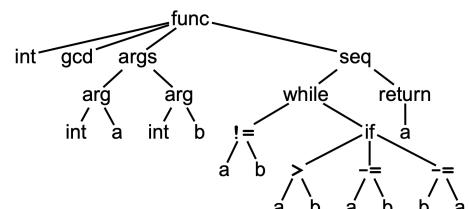
- Introducció
- Visió general
- Anàlisi lèxica
- **Anàlisi sintàctica**
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

51 / 87

Anàlisi sintàctica

L'objectiu de l'analitzador sintàctic (o parser) és convertir una seqüència de tokens en un arbre de sintaxi abstracta que capturi la jerarquia de les construccions.

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```



→ Es descarta informació no rellevant com les paraules clau, els separadors, els parèntesis i els blocs.

→ Es facilita la feina dels propers estadis.

52 / 87

Gramàtiques

La majoria dels LPs es descriuen a través de gramàtiques incontextuals, usant notació BNF (Backus–Naur form).

pgma → expr ; pgma
| ε

expr → expr + expr
| expr - expr
| expr * expr
| expr / expr
| (expr)
| NUM

Les gramàtiques incontextuals permeten descriure llenguatges més amplis que els llenguatges regulars perquè són "recursives".

Exemple: Llenguatge dels mots capicues

- gramàtica incontextual
- expressió regular

→ La recursivitat permet definir jerarquies i niuar elements (parèntesis o blocs).

53 / 87

Gramàtiques

Exemple: Gramàtica de C 

```
translation-unit      : {external-declaration}*
external-declaration   : function-definition
                        | declaration

function-definition    : {declaration-specifier}* declarator {declaration}* compound-statement
declaration-specifier  : storage-class-specifier
                        | type-specifier
                        | type-qualifier

storage-class-specifier : auto
                        | register
                        | static
                        | extern
                        | typedef

type-specifier          : void
                        | char
                        | short
                        | int
                        | long
                        | float
                        | double
                        | signed
                        | unsigned
                        | struct-or-union-specifier
                        | enum specifier
```

54 / 87

Dificultats

- Gramàtiques ambigües
- Prioritat dels operadors
- Associativitat dels operadors
- Analitzadors top-down *vs.* bottom-up
- Recursivitat per la dreta / per l'esquerra

55 / 87

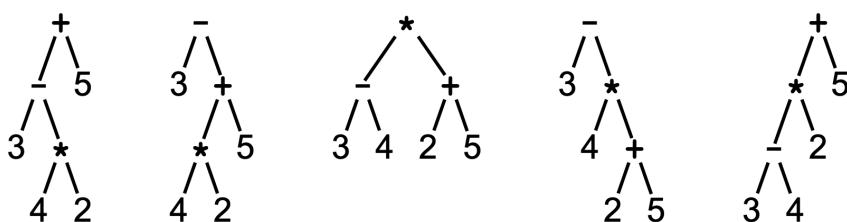
Gramàtiques ambigües

Una gramàtica és ambigua si un mateix text es pot derivar (organitzar en un arbre segons la gramàtica) de diferents maneres.

Per exemple, amb

expr → expr + expr | expr - expr | expr * expr | NUM

el fragment 3 - 4 * 2 + 5 es pot derivar d'aquestes maneres:



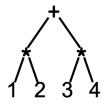
56 / 87

Gramàtiques ambigües

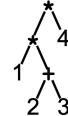
La prioritat i associativitat permet eliminar ambigüitats.

- Prioritat: $1 * 2 + 3 * 4$

* té més prioritat que +

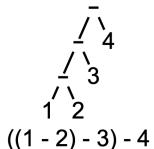


+ té més prioritat que *

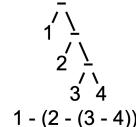


- Associativitat: $1 - 2 - 3 - 4$

associativitat per l'esquerra



associativitat per la dreta



57 / 87

Desambiguació de gramàtiques

Comencem amb:

```
expr → expr + expr  
      | expr - expr  
      | expr * expr  
      | expr / expr  
      | NUM
```

➡ És ambigua: no hi ha la prioritat ni associativitat.

58 / 87

Desambiguació de gramàtiques

Podem assignar prioritats trencant en vàries regles, una per nivell:

```
expr → expr + expr  
      | expr - expr  
      | expr * expr  
      | expr / expr  
      | NUM
```



```
expr → expr + expr  
      | expr - expr  
      | term
```

```
term → term * term  
      | term / term  
      | NUM
```

- ➡ Encara és ambigua: falta associativitat.

59 / 87

Desambiguació de gramàtiques

Podem fer que un costat o altre afecti el següent nivell de prioritat:

```
expr → expr + expr  
      | expr - expr  
      | term
```

```
term → term * term  
      | term / term  
      | NUM
```



```
expr → expr + term  
      | expr - term  
      | term
```

```
term → term * NUM  
      | term / NUM  
      | NUM
```

- ✓ La gramàtica ja no és ambigua.

- ✗ Però el - queda associat per la dreta...

60 / 87

Generadors d'analitzadors sintàctics

Existeixen diferents tècniques per generar analitzadors sintàctics, amb propietats diferents.

- Analitzadors descendents (*top-down parsers*): reconeixen l'entrada d'esquerra a dreta tot buscant derivacions per l'esquerra expandint la gramàtica de l'arrel cap a les fulles.
- Analitzadors ascents (*bottom-up parsers*): reconeixen primer sobre les unitats més petites de l'entrada analitzada abans de reconèixer l'estructura sintàctica segons la gramàtica.

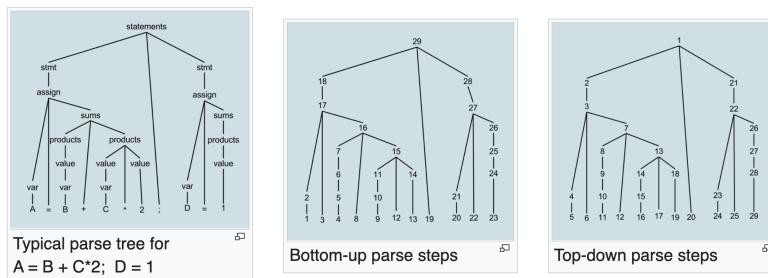


Figura: Wikipedia
61 / 87

Generadors d'analitzadors sintàctics

Analitzadors descendents $LL(k)$

- LL: Left-to-right, Left-most derivation
- k : nombre de tokens que mira endavant

Idea bàsica: mirar el següent token per poder decidir quina producció utilitzar.

Implementació: Associar una funció a cada construcció del LP.

- Si la construcció està definida per una única regla:

La seva funció associada cridarà a les funcions associades a les construccions que apareixen en aquesta regla i comprovarà que els tokens que apareixen en la definició son els que apareixen en la seqüència donada.

- Si la construcció està definida per diverses regles:

La decisió sobre quina regla s'ha d'aplicar es basa en mirar els següents k tokens.

Generadors d'analitzadors sintàctics

Analitzadors descendents LL(1)

Exemple de gramàtica:

```
root    : stmt* 'end'  
        ;  
  
stmt   : 'if' expr 'then' stmt  
        | 'while' expr 'do' stmt  
        | expr ':=' expr  
        ;  
  
expr   : NUMBER  
        | '(' expr ')'  
        ;
```

Exercici:

Implementeu `expr()` i `root()`.

Parser LL(1):

```
def stmt():  
    if current_token() == IF:  
        match(IF)  
        cond = expr()  
        match(THEN)  
        then = stmt()  
        return Node(IF, cond, then)  
    elif current_token() == WHILE:  
        match(WHILE)  
        cond = expr()  
        match(DO)  
        loop = stmt()  
        return Node(WHILE, cond, loop)  
    elif current_token() in [NUMBER, LPAREN]:  
        lvalue = expr()  
        match(ASSIGN)  
        rvalue = expr()  
        return Node(ASSIGN, lvalue, rvalue)  
    else:  
        SyntaxError()
```

63 / 87

Generadors d'analitzadors sintàctics

Analitzadors descendents LL(1)

Inconvenients principals:

- Les produccions no poden tenir prefixos comuns (no sabria quina triar).

$$\begin{aligned} \text{expr} \rightarrow & \text{ ID } (\text{ expr }) \\ | & \text{ ID } = \text{ expr } \end{aligned}$$

- Les regles no poden tenir recursivitat per l'esquerra (es penjaria).

$$\begin{aligned} \text{expr} \rightarrow & \text{ expr } + \text{ term } \\ | & \text{ term } \end{aligned}$$

64 / 87

Generadors d'analitzadors sintàctics

Analitzadors descendents LL(1)

Comencem amb:

```
expr → expr '+' term  
      | expr '-' term  
      | term
```

💡 prefixos comuns

⬇️ factoritzem els prefixos comuns

```
expr → expr ('+' term | '-' term)  
      | term
```

💡 recursivitat per l'esquerra

⬇️ substituim recursivitat per l'esquerra per recursivitat per la dreta

```
expr → term expr2  
expr2 → '+' term expr2  
       | '-' term expr2  
       | ε
```



65 / 87

ANTLR

ANTLR és un analitzador descendente LL(k).
També permet usar * i + a les regles sintàctiques.

```
expr → expr '+' term  
      | expr '-' term  
      | term
```

⬇️ s'escriu senzillament

```
expr : term ('+' term | '-' term) * ;
```

A més, la prioritat dels operadors ve donada per l'ordre d'escriptura:

```
expr : expr '*' expr  
      | expr '+' expr  
      | NUM ;
```

I es pot definir fàcilment l'associativitat:

```
expr : <assoc=right> expr '^' expr  
      | NUM ;
```

66 / 87

Exercicis

P1: Escriviu gramàtiques no ambigües pels llenguatges següents:

1. El conjunt de tots els mots amb as i bs que són palíndroms.
2. Mots que tenen el patró a*b* amb més as que bs.
3. Textos amb parèntesis i claudàtors ben aniuats.
Exemple: ([[] (() [()] [])]).
4. El conjunt de tots els mots amb as i bs tals que a cada a li segueix immediatament per, almenys, una b.
5. El conjunt de tots els mots amb as i bs amb el mateix nombre d'as que bs.
6. El conjunt de tots els mots amb as i bs amb un nombre diferent d'as que bs.
7. Blocs d'instruccions separades per ; a la Pascal. Exemple:
`BEGIN instrucció ; BEGIN instrucció ; instrucció END ; instrucció END.`
8. Blocs d'instruccions acabades per ; a la C.
Exemple: { instrucció ; { instrucció ; instrucció ; } ; instrucció ; }.

P2: Especifiqueu les gramàtiques anteriors amb notació ANTLR, modificant la gramàtica si és necessari.

67 / 87

Exercicis

P3: Sense utilitzar cap eina ni llibreria (ni eval!), escriviu en Haskell, Python o C++ un analitzador descendente LL(1) que llegeixi una seqüència d'expressions i escrigui el resultat de cadascuna d'elles.
[TBD: problema pel Jutge! 😊]

- Entrada:

```
2
2 * 3
2 * 3 + 1
2 * (3 + 1)
10 - 3 - 2
13 / 3
```

- Sortida:

```
2
6
7
8
5
4
```

68 / 87

Exercicis

P4: Sense utilitzar cap eina ni llibreria, escriu en Haskell, Python o C++ un analitzador descentent LL(1) que llegeixi una seqüència d'expressions i construeixi i escrigui l'arbre de sintaxi abstracta de cadascuna. [TBD: problema pel Jutge! 😊]

- Entrada:

`2 * (3 + 1)
(10 - 3 - 2) / 4`

- Sortida:

`(MUL 2 (SUM 3 1))
(DIV (SUB (SUB 10 3) 2) 4)`

69 / 87

Contingut

- Introducció
- Visió general
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

70 / 87

Accions

- Amb un analitzador descendente, es poden executar accions durant el reconeixement de les regles.

Les accions poden aparèixer en qualsevol punt de la regla:

```
regla  :      { /* abans */    }
            regla1
            { /* durant */   }
            regla2
            { /* durant */   }
            regla3
            { /* després */  }
;
```

- La gramàtica esdevé "imperativa".
- Les accions s'entrellacen amb la gramàtica.
- És fàcil entendre què passa i quan passa.

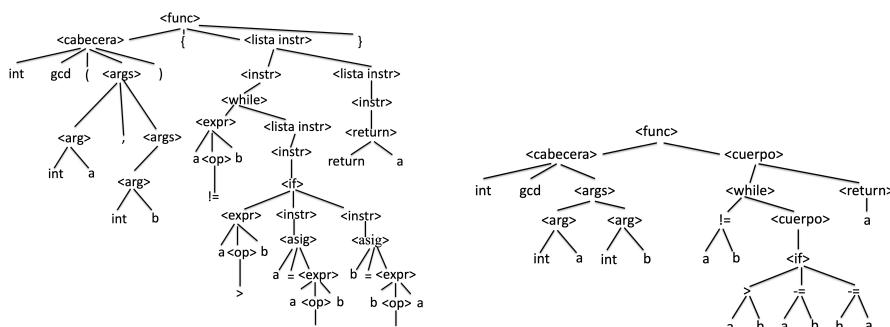
- Amb un analitzador ascendent, només es poden executar accions després de reconèixer una regla.

71 / 87

Arbres de sintaxi

Usualment, les accions construeixen un arbre de sintaxi concreta que segueix les regles de la gramàtica.

Aquest arbre es sol convertir en un arbre de sintaxi abstracta.



- Es separa l'anàlisi de la traducció.
- Es facilita les modificacions tot minimitzant les interaccions.
- Es permet que diferents parts del programa s'analitzin en ordres diferents.

72 / 87

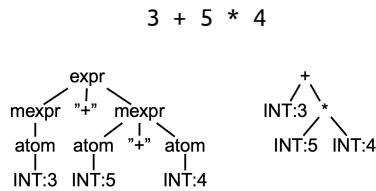
Arbres de sintaxi concreta vs abstracta

Arbre de sintaxi concreta / de derivació: Reflecteix exactament les regles sintàctiques.

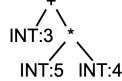
Arbre de sintaxi abstracta (*abstract syntax tree, AST*): Representa el programa fidelment, però elimina i simplifica detalls sintàctics irrelevants.

Exemple: Eliminar regles per desambiguar gramàtica.

```
expr      : mexpr ('+' mexpr) * ;  
mexpr    : atom ('*' atom ) * ;  
atom     : NUM ;
```



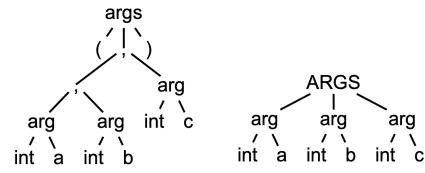
$3 + 5 * 4$



Abstract Syntax Tree

Exemple: Aplanar una llista de paràmetres.

```
int gcd(int a, int b, int c)
```



73 / 87

Ús dels ASTs

Un cop construït l'AST, les etapes següents el recorren per a dur a terme les seves tasques:

- L'anàlisi semàntica verificarà l'ús correcte dels elements del programa.
- El generador de codi visitarà l'arbre i li aplicarà regles per generar codi intermedi.
- L'intèrpret es passejarà per l'arbre per dur a termes les seves instruccions.

74 / 87

ASTs en ANTLR

A partir de la gramàtica, ANTLR pot generar un analitzador descendental amb accions que construeixin un AST.

L'AST es pot visitar a través de *visitors* (un patró de disseny).

ANTLR també genera la interfície dels visitadors, tot generant un esquelet de mètodes que podem heretar.

Cada mètode s'aplica sobre un tipus de node que correspon a cada regla de la gramàtica.

75 / 87

Contingut

- Introducció
- Visió general
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- [Anàlisi semàntica](#)
- Interpretació

76 / 87

Anàlisi semàntica

L'analitzador semàntic recorre l'AST, per obtenir tota la informació necessària per poder generar codi.

Objectius:

- Comprovar la correcció semàntica del programa (comprovació de tipus).
- Resoldre ambigüitats.
- Assignar memòria.
- Construir la taula de símbols.

➡ En aquest curs no entrem en aquest vast tema, que deixem per l'assignatura de Compiladors (recomanada!). Però al tema "Inferència de tipus" veure com fer comprovació de tipus.

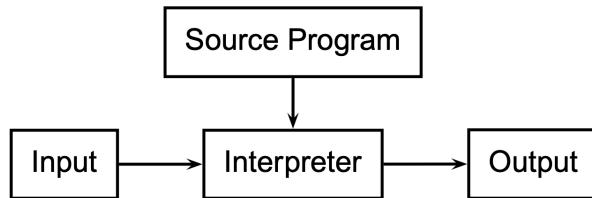
77 / 87

Contingut

- Introducció
- Visió general
- Anàlisi lèxica
- Anàlisi sintàctica
- Arbres de sintaxi abstracta
- Anàlisi semàntica
- Interpretació

78 / 87

Interpretació



Tutorial: Escriure (en Haskell) un intèrpret per a l'AST d'un senzill llenguatge de programació (SimpleLP™).

79 / 87

SimpleLP

Definició del llenguatge

- Tipus de dades: enteros.
- Variables: siempre visibles, inicializadas a 0.
- Operadores aritméticos: + i -.
- Operadores relacionales: ≠ i < (retornen 0 o 1).
- Instrucciones: asignación, composición secuencial, condicional y iteración.

Exemple

```
# Algorisme d'Euclides per calcular el mcd de 105 i 252.
a ← 105
b ← 252
while a ≠ b do
    if a < b then
        b ← b - a
    else
        a ← a - b
    end
end
```

80 / 87

SimpleLP

Tipus de dades per l'AST

```
data Expr
= Val Int
| Var String
| Add Expr Expr
| Sub Expr Expr
| Neq Expr Expr
| Lth Expr Expr
-- valor
-- variable
-- suma (+)
-- resta (-)
-- different-de (=)
-- menor-que (<)

data Instr
= Ass String Expr
| Seq [Instr]
| Cond Expr Instr Instr
| Loop Expr Instr
-- assignació
-- composició seqüencial
-- condicional
-- iteració
```

81 / 87

SimpleLP

Tipus de dades per l'AST

Exemple: AST pel programa anterior:

```
Seq [
  (Ass "a" (Val 105))
  ,(Ass "b" (Val 252))
  ,
  (While
    (Neq (Var "a") (Var "b"))
    (Cond
      (Lth (Var "a") (Var "b"))
      (Ass "b" (Sub (Var "b") (Var "a"))))
      (Ass "a" (Sub (Var "a") (Var "b"))))
    )
  )
]
```

82 / 87

SimpleLP

Memòria

Descrivim el valor de les variables a través d'una memòria de tipus `Mem` amb aquestes operacions:

```
-- retorna una memòria buida
empty :: Mem

-- insereix (o canvia si ja hi era) una clau amb el seu valor
update :: Mem -> String -> Int -> Mem

-- consulta el valor d'una clau en una memòria
search :: Mem -> String -> Maybe Int

-- retorna la llista de claus en una memòria
keys :: Mem -> [String]
```

La implementació seria amb qualsevol diccionari (BST, AVL, hashing, ...).
[Si voleu provar-ho, useu `Map` de `Data.Map`.]

83 / 87

SimpleLP

Avaluació de les expressions

```
--- avalia una expressió en un estat de la memòria
eval :: Expr -> Mem -> Int

eval (Val x) m = x
eval (Var v) m =
  case search v m of
    Nothing -> 0
    Just x -> x
eval (Add e1 e2) m = eval' e1 e2 m (+)
eval (Sub e1 e2) m = eval' e1 e2 m (-)
eval (Neq e1 e2) m = b2i $ eval' e1 e2 m (/=)
eval (Lth e1 e2) m = b2i $ eval' e1 e2 m (<)

eval' e1 e2 m op = op (eval e1 m) (eval e2 m)

b2i False = 0
b2i True  = 1
```

84 / 87

SimpleLP

Interpretació de les instruccions

```
--- retorna l'estat final de la memòria després d'executar una instrucció
--- partint d'un estat inicial de la memòria
exec :: Instr -> Mem -> Mem

exec (Ass v e) m = update v (eval e m) m
exec (Seq []) m = m
exec (Seq (i:is)) m = exec (Seq is) (exec i m)
exec (Cond b i1 i2) m = exec (if eval b m /= 0 then i1 else i2) m
exec (Loop b i) m =
  if eval b m /= 0
    then exec (Loop b i) (exec i m)
  else m
```

85 / 87

SimpleLP

Execució del programa

Escriu el valor final de cada variable (en ordre lexicogràfic) després d'executar una instrucció partint d'una memòria buida.

```
run :: Instr -> IO ()
run i = mapM_ printEntry $ sort $ keys m
where
  m = exec i empty
  printEntry k = do
    putStrLn k
    putStrLn ":"
    print $ fromJust $ search k m
```

Execució amb el programa anterior:

```
a : 21
b : 21
```

86 / 87

Exercicis

1. Modifiqueu l'AST i `eval` per tenir operadors de resta i producte.
2. Modifiqueu l'AST, `exec` i `run` per tal d'afegir a SimpleLP una nova instrucció `print` que escrigui el contingut d'una expressió. Ara `run` només ha d'executar el programa donat partint d'una memòria buida.
3. Afegiu una expressió `read` a SimpleLP que retorni el valor del següent enter de l'entrada.
4. Afegiu una instrucció del tipus `for i ← a .. b`.
5. Feu que `print` pugui escriure una llista de valors (`print a, b, a + b` per ex).
6. Escriviu l'AST d'aquest programa:

```
# factorial en SimpleLP
n ← read
f ← 1
for i ← 2 .. n
    f ← f * i
end
print n, f
```

Llenguatges de Programació

Programació en Python



Jordi Petit, Albert Rubio i Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 2

Contingut

- Sessió 1: **Python bàsic i funcional**
Elements bàsics. Part funcional. Exercicis.
- Sessió 2: **Python avançat**
Classes. Tipus algebraics. Lazyness. Exercicis.
- Sessió 3: **Compiladors**
Compiladors amb ANTLR. Exercicis.

2 / 2

Llenguatges de Programació

Sessió 1: Python bàsic i funcional



Gerard Escudero i Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 27

Contingut

- Elements bàsics
- Iterables
- Part funcional
- Exercicis

2 / 27

Introducció

Paradigmes:

- imperatiu,
- orientat a objectes,
- funcional.

Característiques:

- interpretat,
- llegibilitat,
- *lazyness*.

Té una gran quantitat de llibreries disponibles.

3 / 27

Entorns

'Hello world!'

```
print('Hello world!')  
  
nom = input('Com et dius? ')  
print('Hola', nom + '!')
```

Línia de comandes:

```
python3 script.py
```

idle: entorn Python 3

```
idle o idle3
```

Codificació:

Utilitza 'utf-8'

4 / 27

Blocs i comentaris

Comentaris

```
# això és un comentari
```

Blocs

Els blocs (*suites*) es marquen per l'identació.

```
if condicio:  
    print('ok!')
```

L'estil estàndard *PEP8* es marcar-ho amb 4 espais.

Línies llargues

Podem tallar línies amb \:

```
total = x*100 + \  
       y*10
```

5 / 27

Variabes i assignació

Declaració implícita (valor):

```
a = 2
```

Assignació:

```
x = y = z = 0  
x, y, z = 0, 5, "Llenguatges"  
x, y = y, x
```

Assignació augmentada:

```
a += 3
```

⚠️ No té ni -- ni ++!

6 / 27

Tipus estàndard

Noms

`int, float, complex`

Són tipus i funcions de conversió.

`int` no té rang. Pot tractar nombres arbitràriament llargs.

Operadors usuals excepte: `**` (potència) i `//` (divisió entera)

Booleans

`True, False`

Operadors: `and, or, not`

Funcions de tipus

`type(3)` ➔ `int`

`isinstance(3, int)` ➔ `True`

`isinstance(3, (float, bool))` ➔ `False`

7 / 27

Condicionals

Acció (*statement*):

```
if x < 0:  
    signe = -1  
elif x > 0:  
    signe = 1  
else:  
    signe = 0
```

Els `else` i `elif` són opcionals.

Expressió:

```
x = 'parell' if 5 % 2 == 0 else 'senar'
```

8 / 27

Iteracions

Taula de multiplicar:

while

```
n, i = int(input('n? ')), 1
while i <= 10:
    print(n, 'x', i, '=', n * i)
    i += 1
```

for

```
n = int(input('n? '))
for i in range(1, 11):
    print(n, 'x', i, '=', n * i)
```

El **for** funciona sobre tipus *iterables*.

També podem usar el **break** i el **continue**, amb la semàntica usual sobre tots dos bucles.

9 / 27

Funcions I

Declaració:

```
def primer(n):
    for d in range(2, n // 2 + 1):
        if n % d == 0:
            return False
    return True
```

Crida:

```
primer(5) ➔ True
```

Retorn de múltiples valors:

```
def divisio(a, b):
    return a // b, a % b

x, y = divisio(7, 2) # x ➔ 3, y ➔ 1
```

Quan tornem més d'un valor ho fa internament en forma de *tupla*.

10 / 27

Funcions II

Valors per defecte:

```
from string import punctuation

def remPunc(s, tl=True):
    rt = ''
    for c in s.lower() if tl else s:
        if c not in punctuation:
            rt = rt + c
    return rt

remPunc('Hola, sóc un exemple!')
👉 'hola sóc un exemple'

remPunc('Hola, sóc un exemple!', tl=False)
👉 'Hola sóc un exemple'
```

11 / 27

Contingut

- Elements bàsics
- Iterables
- Part funcional
- Exercicis

12 / 27

Strings I

Tipus str.

Els *strings* són iterables.

Operacions:

```
z = 'Llenguatges'  
z[2]    ↪ 'e'      # posició  
z[3:6]  ↪ 'ngu'    # subcadena  
z[:4]   ↪ 'Llen'    # prefix  
z[8:]  ↪ 'ges'    # sufix  
z + ' Programació' ↪ 'Llenguatges Programació' # concatenar  
z * 3   ↪ 'LlenguatgesLlenguatgesLlenguatges' # repetir  
len(z) ↪ 11       # mida
```

13 / 27

Strings II

Altres operacions i mètodes

```
z = 'Llenguatges'  
'ng' in z ↪ True  
'ng' not in z ↪ False  
z.find('ng') ↪ 3           # cerca i torna posició  
z.count('e') ↪ 2           # comptar  
'Hello world!\n'.strip() ↪ 'Hello world!' # treu el \n  
'1,2,3'.split(',') ↪ ['1', '2', '3'] # parteix un string  
'.'.join(['1', '2', '3']) ↪ '1,2,3' # operació inversa
```

Els *strings* són *immutables*:

```
z[0] = 'l' ✗ # TypeError: 'str' object does not support item assignment
```

14 / 27

Llistes

Les llistes (`list`) són heterogènies:

```
z = ["holà", 5, "llenguatge", 6.63, 2]
```

Tenen les mateixes operacions que els *strings* i també són *iterables*.

Per recórrer dos o més iterables podem utilitzar el `zip`:

```
def prodEscalar(v, w):
    res = 0
    for x, y in zip(v, w):
        res += x * y
    return res
```

Altres operacions predefinides de la classe `list`:

- `append`, `count`, `pop`, etc.

15 / 27

Tuples

Les tuples (`tuple`) són:

- com les llistes
- *immutable*s (de només de lectura)

```
z = ("holà", 5, "llenguatge", 6.63, 2)
z = (5,)      # tupla d'un sol element
              # (5) és l'enter 5
```

Conjunts

Els conjunts (`set`) admeten les operacions:

- `len`, `in`, `not in`, `issubset (<=)`, `issuperset (>=)`,
- `union (|)`, `intersection (&)`, `difference (-)`,
- `add`, `remove` ...

16 / 27

Diccionaris

Els diccionaris (dict):

- contenen parells clau-valor
- permeten accés directe

```
dic = {}                      # diccionari buit
dic["prim"] = "el primer"      # afegir o actualitzar un element
del(dic['prim'])              # esborrar-lo
dic = {"nom": "albert", "num":37899, "dept": "computer science"}  
# inicialitzar-lo amb dades
```

Els diccionaris són iterables (en iterar amb el for recorrem les claus):

```
def suma(d):
    s = 0
    for k in d:
        s += d[k]
    return s

suma({'a': 1, 'b': 2}) ➔ 3
```

17 / 27

Contingut

- Elements bàsics
- Iterables
- Part funcional
- Exercicis

18 / 27

Funcions anònimes

Les funcions són un tipus intern en python (*function*). Poden ser tractades com a dades i, per tant, com a paràmetres d'una funció.

Disposem de funcions anònimes tipus *lambda*:

```
lambda parametres: expressió
```

on els **parametres** són zero o més paràmetres separats per comes.

```
doble = lambda x: 2 * x      # una altra forma de definir funcions  
doble(3)  ↪ 6  
type(doble)  ↪ <class 'function'>
```

Una aplicació pràctica és el paràmetre **key** de les funcions **max**, **min** i **sort**:

```
d = {'a': 2, 'b': 1}  
max(d, key = lambda x: d[x])  ↪ 'a'      # clau amb valor màxim d'un diccionari
```

19 / 27

Funcions d'ordre superior I

map

map(funció, iterable) ↪ generador: aplica la funció a cadascun dels elements de l'iterable.

```
list(map(lambda x: x * 2, [1, 2, 3]))  ↪ [2, 4, 6]
```

filter

filter(funció, iterable) ↪ generador: és el subitable amb els elements que fan certa la funció booleana.

```
mg3 = lambda x: x > 3  
list(filter(mg3, [3, 6, 8, 1]))  ↪ [6, 8]
```

20 / 27

Funcions d'ordre superior II

reduce (fold)

`reduce(funció, iterable[, valor_inicial])` ↗ valor: desplega una funció per l'esquerra.

```
from functools import reduce  
  
reduce(lambda acc,y: acc+y, [3,6,8,1]) ↗ 18  
reduce(lambda acc,y: acc+y, [3,6,8,1], 0) ↗ 18
```

21 / 27

Llistes per comprensió I

amb llistes

[expressió for variable in iterable if expressió]

```
[x ** 2 for x in range(4)] ↗ [0, 1, 4, 9]  
[x for x in [0, 1, 4, 9] if x % 2 == 0] ↗ [0, 4]  
[(x, y) for x in [1, 2] for y in 'ab']  
↗ [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

amb conjunts

```
{x for x in range(4) if x % 2 == 0}  
↗ {0, 2}
```

22 / 27

Llistes per comprensió II

amb diccionaris

```
{x: x % 2 == 0 for x in range(4)}  
👉 {0: True, 1: False, 2: True, 3: False}
```

amb generadors

```
from itertools import count      # count és un generador infinit  
  
g = (x**2 for x in count(1))    # g és un generador  
                                # dels quadrats dels naturals  
  
next(g) 👉 1  
  
[next(g) for _ in range(4)] 👉 [4, 9, 16, 25]
```

23 / 27

Mòdul *operator*

La llibreria *operator* conté tots els operadors estàndard en forma de funcions, per a ser usades en funcions d'ordre superior.

```
from operator import mul  
from functools import reduce  
  
factorial = lambda n: reduce(mul, range(1, n+1))  
  
factorial(5) 👉 120
```

Alguns exemples de funcions que conté són:

- `iadd(a, b)`: equivalent a `a += b`
- `attrgetter(attr)`:

```
f = attrgetter('name')  
f(b) 👉 b.name
```

24 / 27

Mòdul *itertools*

Aquesta llibreria conté moltes funcions relacionades amb les iteracions.

Té moltes funcions equivalents a Haskell:

```
from operator import mul
from itertools import accumulate

factorials = lambda n: accumulate(range(1, n + 1), mul)
[x for x in factorials(5)] ➔ [1, 2, 6, 24, 120]
```

Altres funcions amb equivalents Haskell són: `dropwhile`, `islice` (`take`), `repeat` o `takewhile`.

Té algunes funcions que fan d'iteradors combinatoris: `product`, `permutations`, `combinations` o `combinations_with_replacement`.

És interessant fer un repàs a la documentació d'aquesta llibreria: <https://docs.python.org/3.7/library/itertools.html>

25 / 27

Contingut

- Elements bàsics
- Iterables
- Part funcional
- Exercicis

26 / 27

Exercicis

- Feu aquests problemes de Jutge.org:
 - [P84591](#) Funcions amb nombres
 - [P51956](#) Funcions amb llistes
 - [P80049](#) Ús d'iterables
 - [P66679](#) Llistes per comprensió
 - [P73993](#) Funcions d'ordre superior

Llenguatges de Programació

Sessió 2: Python avançat



Gerard Escudero i Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 17

Contingut

- Classes
- Tipus algebraics
- *Lazyness*
- Exercicis

2 / 17

Classes I

Exemple de classe:

```
class Treballador:  
    treCompt = 0          # atribut de classe:  
                        # un únic valor per classe  
    def __init__(self, nom, salari): # constructor  
        self.nom = nom          # definició d'atribut  
        self.salari = salari    # self representa el objecte creat  
        Treballador.treCompt += 1  
  
    def getCompt(self):        # definició de mètode  
        return Treballador.treCompt  
  
    def getSalari(self):  
        return self.salari
```

Exemple d'interacció:

```
tre1 = Treballador("Pep", 2000)  
tre2 = Treballador("Joan", 2500)  
tre1.getSalari() ➔ 2000  
tre1.salari ➔ 2000  
tre1.getCompt() ➔ 2  
Treballador.treCompt ➔ 2
```

3 / 17

Classes II

Podem afegir, eliminar o modificar atributs de classes i objectes en qualsevol moment:

```
tre1.edat = 8  
tre1.edat ➔ 8  
hasattr(tre1, 'edat') ➔ True  
del tre1.edat  
hasattr(tre1, 'edat') ➔ False
```

La comanda `dir` ens retorna la llista d'atributs d'un objecte. Hi ha molts predefinits:

- `__dict__`, `__doc__`, `__name__`, `__module__`, `__bases__`.

Atributs ocells

```
class Treballador:  
    __treCompt = 0          # atribut ocult: començant per 2 _  
  
    tre1.__treCompt ➗  
    tre1.tre1._Treballador__treCompt ➔ 2
```

4 / 17

Herència

Exemple:

```
class Fill(Treballador):    # pare entre ( )
    def fillMetode(self):
        print('Cridem al metode del fill')

tre3 = Fill('Manel', 1000)
tre3.fillMetode()
```

Podem tenir herència múltiple:

```
class A: # definim la classe A
.....
class B: # definim la classe B
.....
class C(A, B): # subclasse de A i B
.....
```

Per fer comprovacions:

- `issubclass(sub, sup)`
- `isinstance(obj, Class)`

5 / 17

Contingut

- Classes
- Tipus algebraics
- *Lazyness*
- Exercicis

6 / 17

Tipus enumerats

Donen la llista de valors possibles dels objectes:

Pedra, paper, tisores:

```
class Pedra:  
    pass  
  
class Paper:  
    pass  
  
class Tisores:  
    pass  
  
Jugada = Pedra | Paper | Tisores  
  
guanya = lambda a, b: (a, b) in [(Paper, Pedra), (Pedra, Tisores), (Tisores, Paper)]
```

Exemple:

Guanya la primera a la segona?

guanya(Paper, Pedra) ➔ True

7 / 17

Tipus algebraics

Defineixen constructors amb zero o més dades associades:

```
from dataclasses import dataclass  
  
@dataclass  
class Rectangle:  
    amplada: float  
    alçada: float  
  
@dataclass  
class Quadrat:  
    mida: float  
  
@dataclass  
class Cercle:  
    radi: float  
  
class Punt:  
    pass  
  
Forma = Rectangle | Quadrat | Cercle | Punt
```

- El decorador `dataclass` defineix automàticament mètodes com `__init__`.

8 / 17

Funcions amb tipus algebraics

Declaració:

```
from math import pi

def area(f):
    match f:
        case Rectangle(amplada, alçada):
            return amplada * alçada
        case Quadrat(mida):
            return area(Rectangle(mida, mida))
        case Cercle(radi):
            return pi * radi**2
        case Punt():
            return 0
```

- `match` permet reconèixer patrons

Crida:

```
area(Quadrat(2.0)) ➔ 4.0
```

9 / 17

Tipus recursius

Arbre binari:

```
from __future__ import annotations
from dataclasses import dataclass

class Buit:
    pass

@dataclass
class Node:
    val: int
    esq: Arbre
    dre: Arbre

Arbre = Node | Buit

def mida(a: Arbre) -> int:
    match a:
        case Buit():
            return 0
        case Node(x, e, d):
            return 1 + mida(e) + mida(d)

t = Node(1,Node(2,Buit(),Buit()),Node(3,Buit(),Buit()))
mida(t) ➔ 3
```

- `annotations` permet els tipus recursius (en aquest cas `Arbre`).

10 / 17

Contingut

- Classes
- Tipus algebraics
- *Lazyness*
- Exercicis

11 / 17

Iteradors

En python existeixen mols tipus que són iterables:

- strings, llistes, diccionaris i conjunts
- definits per l'usuari.

El protocol d'iteració el defineixen els mètodes:

- `__iter__`
- `__next__`

i llença l'excepció `StopIteration` en acabar:

```
s = 'abc'  
for c in s:  
    print(c)  
  
a  
b  
c
```

```
s = 'abc'  
it = iter(s)  
while True:  
    try:  
        print(it.__next__())  
    except StopIteration:  
        break
```

12 / 17

Generadors I

Són el mecanisme que permeten l'avaluació *lazy* en python3.

Exemple: sèrie de fibonacci fins a un nombre determinat.

```
def fib(n):
    a = 0
    yield a
    b = 1
    while True:
        if b <= n:
            yield b
            a, b = b, a + b
        else:
            raise StopIteration
    f = fib(1)
    next(f) ➔ 0
    next(f) ➔ 1
    next(f) ➔ 1
    next(f) ➔ StopIteration
    ## La comanda yield para l'execució
    ## de la funció fins a la següent
    ## invocació de next.
    ## amb list comprehension
    [x for x in fib(25)]
    ➔
    [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

13 / 17

Generadors II

Els generadors poden ser infinits:

```
def fib2():
    a = 0
    yield a
    b = 1
    while True:
        yield b
        a, b = b, a + b
    f = fib2()
    [next(f) for _ in range(8)]
    ➔
    [0, 1, 1, 2, 3, 5, 8, 13]
```

14 / 17

Generadors III

classe amb generador:

```
class fib:  
    def __iter__(self):  
        a, b = 0, 1  
        while True:  
            yield b  
            a, b = b, a + b  
  
for (i, x) in enumerate(fib(), 1):  
    if i > 10:  
        break  
    print(i, x)  
  
1 1  
2 1  
3 2  
4 3  
5 5  
6 8
```

classe amb generador i iterador:

```
class fib2:  
    def __init__(self):  
        self.gen = self.__iter__()  
  
    def __next__(self):  
        return next(self.gen)  
  
    def __iter__(self):  
        a, b = 0, 1  
        while True:  
            yield b  
            a, b = b, a + b  
  
f = fib2()  
print([next(f) for _ in range(6)])  
[1, 1, 2, 3, 5, 8]
```

Són útils quan volem tenir varíes instàncies del generador funcionant a l'hora.

15 / 17

Contingut

- Classes
- Tipus algebraics
- *Lazyness*
- Exercicis

16 / 17

Exercicis

- Feu aquests problemes de Jutge.org:
 - [P71608](#) Classe per arbres
 - [P45231](#) Generadors
 - [P53498](#) Definició d'un iterable

Llenguatges de Programació

Sessió 3: compiladors



Gerard Escudero i Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



1 / 27

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- Exercicis
- Taules de símbols
- Tractament d'errors
- Referències

2 / 27

Instal·lació de l'ANTLR4 (per Python)

Requeriments:

- Python 3

Instruccions:

```
pip install antlr4-tools  
antlr4  
  
pip install antlr4-python3-runtime
```

Windows: s'ha de fer alguna cosa més, seguiu la referència.

- [antlr4-tools reference](#)

3 / 27

Contingut

- ANTLR
- [Gramàtica](#)
- *Visitor*
- Exercicis
- Taules de símbols
- Tractament d'errors
- Referències

4 / 27

El primer programa ANTLR

Arxiu de gramàtica `exprs.g4`:

```
// Gramàtica per expressions senzilles

grammar exprs;

root : expr           // l'etiqueta ja és root
      ;

expr : expr '+' expr    # suma
      | NUM            # numero
      ;

NUM : [0-9]+ ;
WS  : [ \t\n\r]+ -> skip ;
```

⚠ Noteu que el nom de l'arxiu ha de concordar amb el de la gramàtica.

- `expr`: definició de la gramàtica per la suma de nombres naturals.
- `skip`: indica a l'escàner que el token WS no ha d'arribar al parser.
- #: etiqueta per diferenciar branques de les regles (no és un comentari!)

5 / 27

Compilació a Python3

La comanda

```
antlr4 -Dlanguage=Python3 -no-listener exprs.g4      # antlr en MacOS
```

genera els arxius:

- `exprsLexer.py` i `exprsLexer.tokens`
- `exprsParser.py` i `exprs.tokens`

⚠ Noteu que els arxius anteriors comencen pel nom de la gramàtica.

6 / 27

Construcció de l'script principal

Script de test exprs.py:

```
from antlr4 import *
from exprsLexer import exprsLexer
from exprsParser import exprsParser

input_stream = InputStream(input('? '))
lexer = exprsLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = exprsParser(token_stream)
tree = parser.root()
print(tree.toStringTree(recog=parser))
```

Noteu que aquest script processa una única línia d'entrada per consola.

En funcionament:

```
3 + 4
  ↗
(root (expr (expr 3) + (expr 4)))
  ↗
line 1:3 missing NUM at '<EOF>'
(root (expr (expr 3) +
  (expr <missing NUM>)))
```

7 / 27

Obtenció de l'entrada

una única línia

```
input_stream = InputStream(input('? '))
```

stdin

```
input_stream = StdinStream()
```

un arxiu passat com a paràmetre

```
input_stream = FileStream(nom_fitxer)
```

arxius amb unicode

```
input_stream = FileStream(nom_fitxer, encoding='utf-8')
```

En aquest cas haurem d'incloure a la gramàtica aquest tipus de caràcters:

```
WORD : [a-zA-Z\u0080-\u00FF]+ ;
```

8 / 27

Notes sobre gramàtiques

Recursivitat per l'esquerra:

Amb les versions anteriors no es podia afegir una regla de l'estil:
`expr : expr '*' expr`

Per solucionar això s'afegien regles tipus `expr : NUM '*' expr`

Precedència d'operadors:

Amb l'ordre d'escriptura:

```
expr : expr '*' expr  
      | expr '+' expr  
      | INT  
      ;
```

Associativitat:

L'associativitat com la potència queda com:

```
expr : <assoc=right> expr '^' expr  
      | INT  
      ;
```

9 / 27

Exercici 1

Afegiu a la gramàtica els operadors de:

- resta
- multiplicació
- divisió
- potència

Tingueu en compte:

- la precedència d'operadors
- l'associativitat a la dreta de la potència

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- Exercicis
- Taules de símbols
- Tractament d'errors
- Referències

11 / 27

Visitors

Els *visitors* són *tree walkers*, un mecanisme per recórrer els ASTs. Amb la comanda:

```
antlr4 -Dlanguage=Python3 -no-listener -visitor exprs.g4      # antlr en MacOS
```

compilarem la gramàtica i generarem la class base del visitador (`exprsVisitor.py`):

```
# Generated from exprs.g4 by ANTLR 4.11.1
from antlr4 import *
if __name__ is not None and "." in __name__
    from .exprsParser import exprsParser
else:
    from exprsParser import exprsParser

# This class defines a complete generic visitor
class exprsVisitor(ParseTreeVisitor):

    # Visit a parse tree produced by exprsParser
    def visitRoot(self, ctx:exprsParser.FactorContext):
        return self.visitChildren(ctx)
    ...
```

visitRoot és el *callback* associat a la regla root per visitar-la.

Quan hi ha una etiqueta com ara `# suma`, la regle és visitSuma.

La crida a `self.visit(node)` visita el visitador associat al tipus de node.

12 / 27

Visitor per recórrer l'arbre

Classe `visitorTreeVisitor.py` per mostrar l'arbre heretant de la classe base:

```
class TreeVisitor(exprsVisitor):

    def __init__(self):
        self.nivell = 0

    def visitSuma(self, ctx):
        [expressio1, operador, expressio2] = list(ctx.getChildren())
        print(' ' * self.nivell + '+')
        self.nivell += 1
        self.visit(expressio1)
        self.visit(expressio2)
        self.nivell -= 1

    def visitNumero(self, ctx):
        [numero] = list(ctx.getChildren())
        print(" " * self.nivell + numero.getText())
```

- Cada funció de visita obté els fills del node `ctx` amb `getChildren()`, i:
 - visita els fills sintàctics amb `self.visit(ctx_i)`, o
 - obté algun atribut dels fills lèxics, com ara el seu text amb `ctx_i.getText()`.

13 / 27

Ús del visitor

L'script l'hem de modificar:

```
from antlr4 import *
from exprsLexer import exprsLexer
from exprsParser import exprsParser
from exprsVisitor import exprsVisitor

class TreeVisitor(exprsVisitor):
    ...

    input_stream = InputStream(input('? '))
    lexer = exprsLexer(input_stream)
    token_stream = CommonTokenStream(lexer)
    parser = exprsParser(token_stream)
    tree = parser.root()

    visitor = TreeVisitor()
    visitor.visit(tree)
```

14 / 27

Execució

Un exemple de resultat de l'script anterior:

```
2 + 3 + 4  
+  
+  
2  
3  
4
```

Exercici 2

Afegiu el mecanisme per mostrar l'arbre generat a la gramàtica de l'exercici 1.

15 / 27

Avaluació i interpretació d'ASTs

Visitor per avaluar les expressions:

```
class EvalVisitor(exprsVisitor):  
  
    def visitRoot(self, ctx):  
        [expressio] = list(ctx.getChildren())  
        print(self.visit(expressio))  
  
    def visitSuma(self, ctx):  
        [expressio1, operador, expressio2] = list(ctx.getChildren())  
        return self.visit(expressio1) + self.visit(expressio2)  
  
    def visitNumero(self, ctx):  
        [numero] = list(ctx.getChildren())  
        return int(numero.getText())
```

Exemple:

3 + 4 + 5 ➔ 12

Nota: podeu utilitzar més d'un visitor en un script.

16 / 27

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- [Exercicis](#)
- Taules de símbols
- Tractament d'errors
- Referències

17 / 27

Exercici 3

Afegiu el tractament d'avaluació per la resta d'operadors de l'exercici 3.

Exercici 4

Definiu una gramàtica i el seu mecanisme d'avaluació/execució per a quelcom tipus:

```
x := 3 + 5
write x
y := 3 + x + 5
write y
```

Nota: es pot utilitzar un diccionari com a taula de símbols.

18 / 27

Exercici 5

Amplieu l'exercici anterior per a que tracti quelcom com el següent:

```
c := 0
b := c + 5
if c = 0 then
    write b
end
```

Exercici 6

Exploreu que passa si realitzem l'exercici anterior sense el token `end`.

Exercici 7

Amplieu l'exercici anterior per a que tracti l'estructura `while`:

```
i := 1
while i <= 11 do
    write i * 2
    i := i + 1
end
```

19 / 27

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- Exercicis
- [Taules de símbols](#)
- Tractament d'errors
- Referències

20 / 27

Què passa amb les funcions?

Imagineu una llenguatge tipus:

```
function sm(x, y)
    return x + y
end

main
    a := 1 + 2
    b := a * 2
    write sm(a, b)
end
```

amb només:

- variables locals
- paràmetres per valor

Qüestions a tenir en compte:

1. La taula de símbols pot ser una *pila de diccionaris*.
2. En *visitar* la declaració de funcions, per a cada funció, hem de guardar en una estructura:
 - El seu nom (*id*)
 - La seva llista de paràmetres (*ids*)
 - El contexte (node de l'AST) del seu bloc de codi (per a poder fer un `self.visit(bloc)` en trobar la crida)
3. S'ha de gestionar el `return` en cascada.

Exercici 8

Amplieu l'exercici anterior per a incloure funcions d'aquest tipus.

21 / 27

Exercici 9

Comproveu que el vostre programa funciona amb recursitat:

```
function fibo(n)
    if n = 0 then
        return 0
    end
    if n = 1 then
        return 1
    end
    return fibo(n-1) + fibo(n-2)
end

main
    a := 1
    while a <> 7 do
        write fibo(a)
        a := a + 1
    end
end
```

22 / 27

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- Exercicis
- Taules de símbols
- **Tractament d'errors**
- Referències

23 / 27

Errors sintàctics

Antlr té un parell mètodes per tractar-los:

- `getNumberOfSyntaxErrors`: ens indica els errors sintàctics
- `removeErrorListeners`: desactiva els missatges de les excepcions del parser

Exemples:

Amb aquest codi evitem cridar el `visitor` si s'ha produït un error sintàctic.

```
if parser.getNumberOfSyntaxErrors() == 0:  
    visitor = TreeVisitor()  
    visitor.visit(tree)  
else:  
    print(parser.getNumberOfSyntaxErrors(), 'errors de sintaxi.')  
    print(tree.toStringTree(recog=parser))
```

Amb aquest codi evitem els missatges de les excepcions que llença el parser.

```
parser = exprsParser(token_stream)  
parser.removeErrorListeners()  
tree = parser.root()
```

24 / 27

Errors lèxics

Es produeixen quan fiquem un símbol no reconegut per la gramàtica.

Per evitar aquests errors hem d'afegir una regla al final de la gramàtica:

```
LEXICAL_ERROR : . ;
```

i desactivar els missatges de les excepcions del lexer:

```
lexer = exprsLexer(input_stream)
lexer.removeErrorListeners()
token_stream = CommonTokenStream(lexer)
```

25 / 27

Contingut

- ANTLR
- Gramàtica
- *Visitor*
- Exercicis
- Taules de símbols
- Tractament d'errors
- Referències

26 / 27

Referències

1. Terence Parr. *The Definitive ANTLR 4 Reference*, 2nd Edition. Pragmatic Bookshelf, 2013.
2. Alan Hohn. *ANTLR4 Python Example*. Últim accés: 26/1/2019. <https://github.com/AlanHohn/antlr4-python>
3. Guillem Godoy i Ramón Ferrer. *Parsing and AST construction with PCCTS*. Materials d'LP, 2011.

Llenguatges de Programació

Inferència de tipus



Jordi Petit, Albert Rubio

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



1 / 45

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

2 / 45

Inferència de tipus

La inferència de tipus és la detecció automàtica dels tipus de les expressions en un llenguatge de programació.

Permet fer més fàcils moltes tasques de programació, sense comprometre la seguretat de la comprovació de tipus.

Té sentit en llenguatges fortament tipats.

És un característica habitual dels llenguatges funcionals.

Alguns LPs amb inferència de tipus:

- C++ >= 11
- Haskell
- C#
- D
- Go
- Java >= 10
- Scala
- ...

3 / 45

Inferència de tipus a C++

La inferència de tipus apareix a la versió 11 de l'estàndard de C++.

- **auto**: Dedueix el tipus d'una variable a través de la seva inicialització:

```
map<int, string> m;
auto x = 12;           // x és un int
auto it = m.find(x);  // x és un map<int, string>::iterator
```

- **decltype**: Obté el tipus d'una expressió.

```
int x = 12;
decltype(x + 1) y = 0;    // y és un int
```

4 / 45

Inferència de tipus a Haskell

- En la majoria de casos no cal definir els tipus.
- Es poden demanar els tipus inferits (que inclouen classes, si cal).

```
λ> :type 3 * 4
👉 3 * 4 :: Num a => a

λ> :type odd (3 * 4)
👉 odd (3 * 4) :: Bool
```

- Algunes situacions estranyes.
 - *Monomorphism restriction*: Sovint no es pot sobrecarregar una funció si no es dona una declaració explícita de tipus.

5 / 45

Inferència de tipus

Problema: Donat un programa, trobar el tipus més general de les seves expressions dins del sistema de tipus del LP.

Solució presentada: Algorisme de Milner.

6 / 45

Contingut

- Introducció
- [Algorisme de Milner](#)
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

7 / 45

Inferència de tipus

Algorisme de Milner

- Curry i Hindley havien desenvolupat idees similars independentment en el context del λ -càcul.
- Hindley–Milner i Damas–Milner
- L'algorisme és similar a la "unificació".

Propietats

- Complet.
- Computa el tipus més general possible sense necessitat d'anotacions.
- Eficient: gairebé lineal (inversa de la funció d'Ackermann). L'eficiència depèn de l'algorisme d'unificació que s'apliqui.

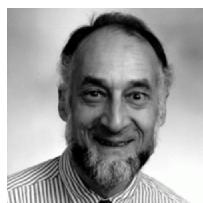


Foto: Domini públic

8 / 45

Algorisme de Milner

Descripció general

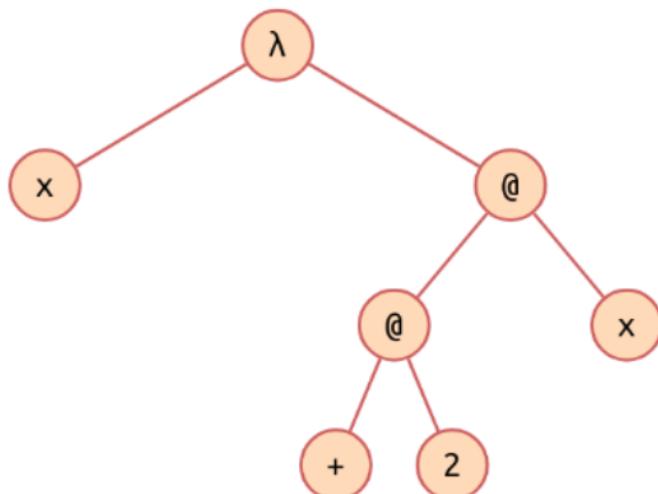
1. Es genera l'arbre de sintaxi de l'expressió (currificant totes les aplicacions).
2. S'etiqueta cada node de l'arbre amb un tipus:
 - Si el tipus és conegut, s'etiqueta amb aquest tipus.
 - Altrament, s'etiqueta amb una nova variable de tipus.
3. Es genera un conjunt de restriccions (d'igualtat principalment) a partir de l'arbre de l'expressió i les operacions que hi intervenen:
 - Aplicació,
 - Abstracció,
 - let, where,
 - case, guardes, patrons,
 - ...
4. Es resolen les restriccions mitjançant unificació.

9 / 45

Primer exemple

$\lambda x \rightarrow (+) 2 x$

Arbre de l'expressió currificada:



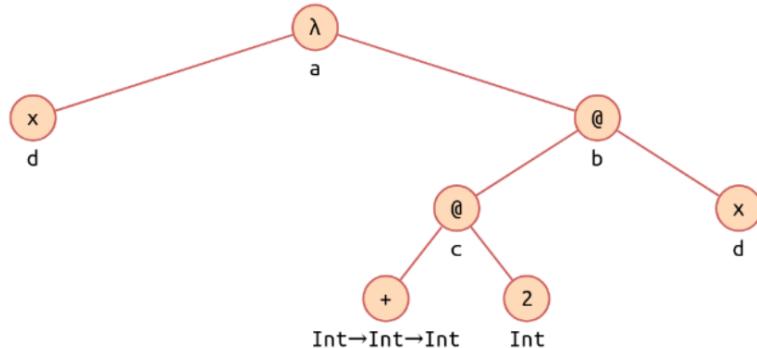
10 / 45

Primer exemple

\x -> (+) 2 x

Etiquetem els nodes:

- Si el tipus és conegut, se'ls etiqueta amb el seu tipus.
- Altrament, se'ls etiqueta amb una nova variable de tipus.
- Nodes iguals han de tenir etiquetes iguals.

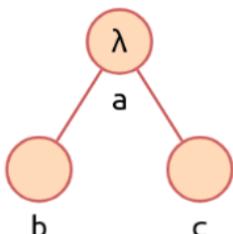


11 / 45

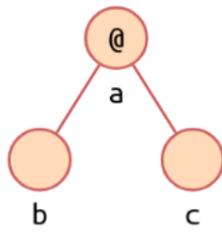
Algorisme de Milner

Regles per generar les equacions

- Abstracció:



- Aplicació:



- Equació: $a = b \rightarrow c$

- Equació: $b = c \rightarrow a$

12 / 45

Primer exemple

- Obtenim les equacions:

$$a = d \rightarrow b$$

$$c = d \rightarrow b$$

$$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} = \text{Int} \rightarrow c$$

- Solucionem les equacions:

$$a = \text{Int} \rightarrow \text{Int}$$

$$b = \text{Int}$$

$$c = \text{Int} \rightarrow \text{Int}$$

$$d = \text{Int}$$

- El tipus de l'expressió és el de l'arrel (a):

$$\lambda x \rightarrow (+) 2 x :: \text{Int} \rightarrow \text{Int}$$

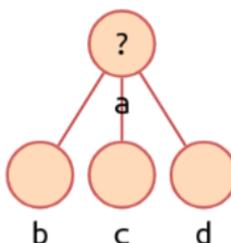
- Recordeu: \rightarrow associa per la dreta: $a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$
- Recordeu: aplicació associa per l'esquerra: $f x y = (f x) y$

13 / 45

Algorisme de Milner

Més regles per generar les equacions

- Condicional if-then-else:



- Equacions:

- $b = \text{Bool}$

- $a = c = d$

Aquesta regla no és estrictament necessària, ja que if-then-else només és una funció genèrica normal de tipus $\text{Bool} \rightarrow a \rightarrow a \rightarrow a$, però estalvia espai.

14 / 45

Segon exemple

`foldl (\a b -> a && b) True xs`

Creeu l'arbre de l'expressió currificada...

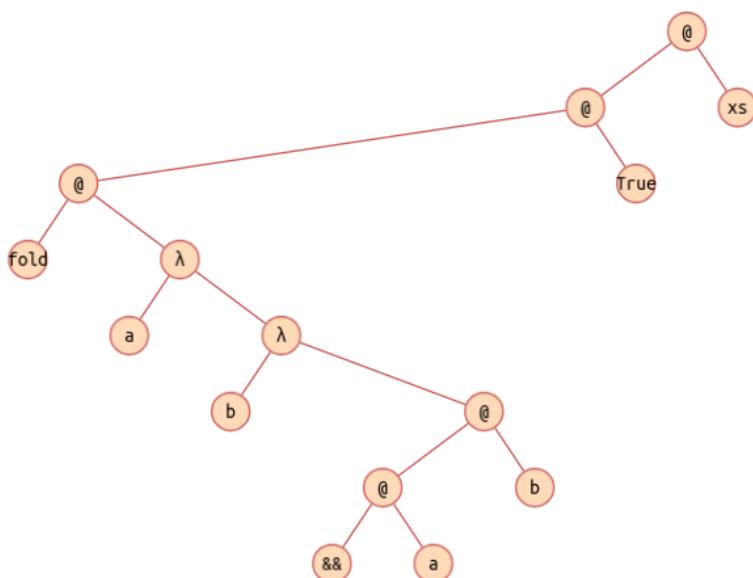


15 / 45

Segon exemple

`foldl (\a b -> a && b) True xs`

Arbre de l'expressió currificada:

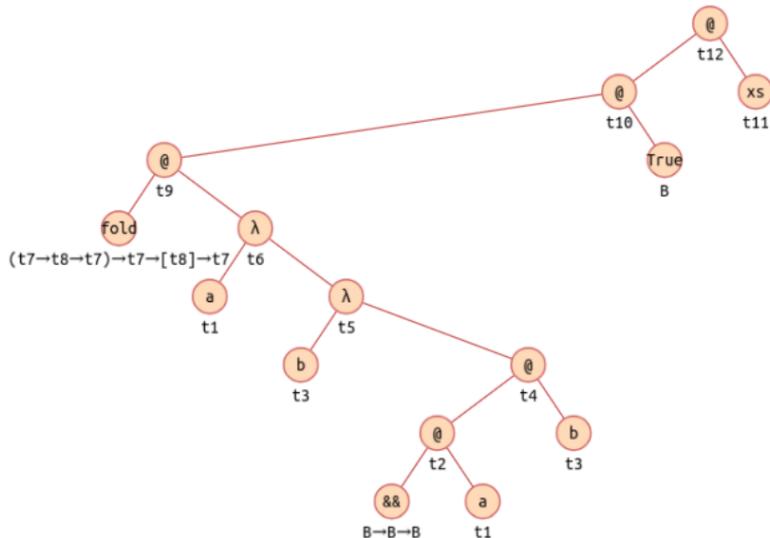


16 / 45

Segon exemple

foldl ($\lambda a b \rightarrow a \&& b$) **True** **xs**

Arbre etiquetat amb tipus: (B és Bool)



17 / 45

Segon exemple — Us toca!

Equacions:

- $t_1 = \text{Bool}$
- $t_2 = \text{Bool} \rightarrow \text{Bool}$
- $t_3 = \text{Bool}$
- $t_4 = \text{Bool}$
- $t_5 = \text{Bool} \rightarrow \text{Bool}$
- $t_6 = \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
- $(t_7 \rightarrow t_8 \rightarrow t_7) \rightarrow t_7 \rightarrow [t_8] \rightarrow t_7 = (\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}) \rightarrow t_9$
- $t_9 = \text{Bool} \rightarrow t_{10}$
- $t_{10} = t_{11} \rightarrow t_{12}$

Solucioneu... (volem t_{12})



18 / 45

Segon exemple

Equacions:

- $t_1 = \text{Bool}$
- $t_2 = \text{Bool} \rightarrow \text{Bool}$
- $t_3 = \text{Bool}$
- $t_4 = \text{Bool}$
- $t_5 = \text{Bool} \rightarrow \text{Bool}$
- $t_6 = \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
- $(t_7 \rightarrow t_8 \rightarrow t_7) \rightarrow t_7 \rightarrow [t_8] \rightarrow t_7 = (\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}) \rightarrow t_9$
- $t_9 = \text{Bool} \rightarrow t_{10}$
- $t_{10} = t_{11} \rightarrow t_{12}$

Solució:

- $t_7 = \text{Bool}$
- $t_8 = \text{Bool}$
- $t_9 = \text{Bool} \rightarrow [\text{Bool}] \rightarrow \text{Bool}$
- $t_{10} = [\text{Bool}] \rightarrow \text{Bool}$
- $t_{11} = [\text{Bool}]$
- $t_{12} = \text{Bool}$ (arrel de l'expressió)

19 / 45

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

20 / 45

Exercicis

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`2 + 3 + 4`

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`2 + 3 <= 2 + 2`

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`map (* 2)`

(Suposeu `(*) :: Int -> Int -> Int`)

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`foldl (flip (:)) []`

21 / 45

Exercicis

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`\f x -> f $ f x`

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`\f -> f . f`

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`\x y -> if y /= 0 then Just (x `div` y) else Nothing`

(Suposeu `div :: Int -> Int -> Int`)

- Utilitzeu l'algorisme de Milner per inferir el tipus de:

`\xs ys -> zipWith (,) xs ys`

22 / 45

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

23 / 45

Definició de funció

```
map f l = if null l then [] else f (head l) : map f (tail l)
```

Podem entendre una definició com una funció que, aplicada als paràmetres, torna la part dreta de la definició:

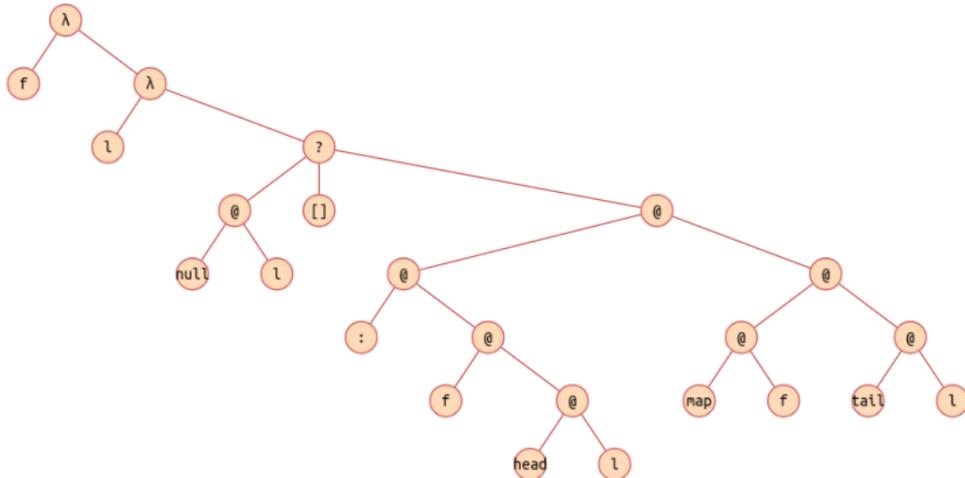
```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

24 / 45

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Arbre de l'expressió:

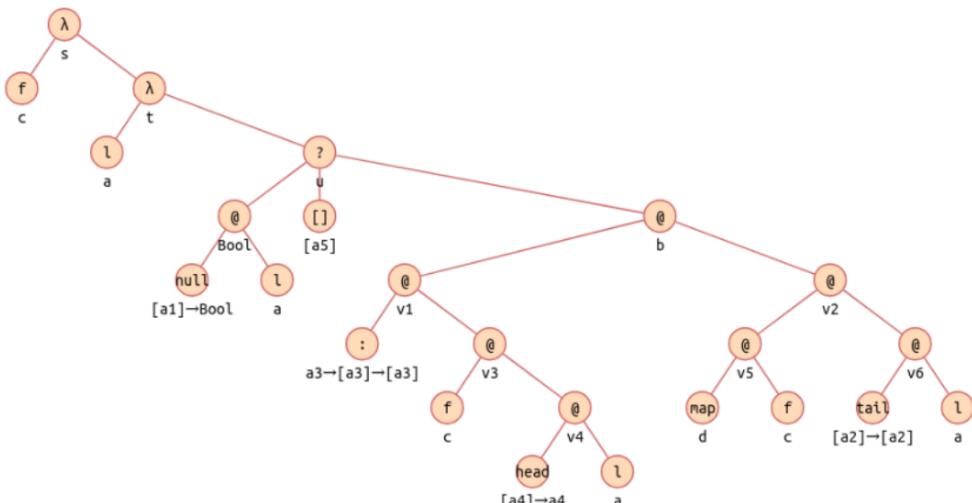


25 / 45

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Arbre etiquetat amb tipus:



26 / 45

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Equacions:

- $s = c \rightarrow t$
 - $t = a \rightarrow u$
 - $u = [a_5]$
 - $u = b$
 - $[a_1] \rightarrow \text{Bool} = a \rightarrow \text{Bool}$
 - $v_1 = v_2 \rightarrow b$
 - $a_3 \rightarrow [a_3] \rightarrow [a_3] = v_3 \rightarrow v_1$
 - $c = v_4 \rightarrow v_3$
 - $[a_4] \rightarrow a_4 = a \rightarrow [v_4]$
 - $v_5 = v_6 \rightarrow v_2$
 - $d = c \rightarrow v_5$
 - $[a_2] \rightarrow [a_2] = a \rightarrow v_6$
- $s = d$ (per establir que el `map` té el mateix tipus a la definició i a l'ús recursiu)

27 / 45

Definició de funció

```
\f -> \l -> if null l then [] else f (head l) : map f (tail l)
```

Solució:

- $a = [a_1]$
- $a_2 = a_1$
- $a_4 = a_1$
- $a_5 = a_3$
- $b = [a_3]$
- $c = a_1 \rightarrow a_3$
- $v_1 = [a_3] \rightarrow [a_3]$
- $v_2 = [a_3]$
- $v_3 = a_3$
- $v_4 = a_1$
- $v_5 = [a_1] \rightarrow [a_3]$
- $v_6 = [a_1]$
- $d = (a_1 \rightarrow a_3) \rightarrow [a_1] \rightarrow [a_3]$
- $s = (a_1 \rightarrow a_3) \rightarrow [a_1] \rightarrow [a_3]$ (arrel)

28 / 45

Definició de funció amb patrons

`map f (x : xs) = f x : map f xs`

En aquest cas la introducció de lambdes és una mica diferent, ja que tractem els patrons com si fossin variables lliures:

`\f -> \(x : xs) -> f x : map f xs`

Noteu que ara hem de considerar que el primer argument de la lambda pot ser una expressió, que tractarem igual que les demés.

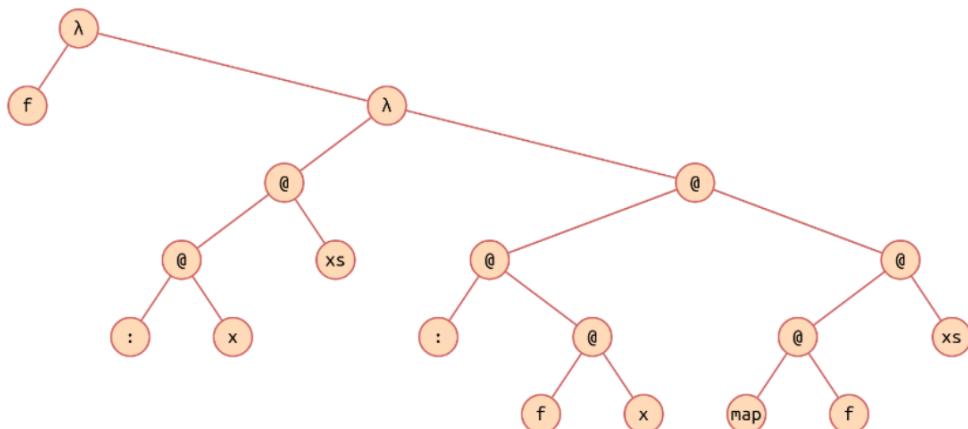
Totes les variables del patró queden lligades per la lambda.

29 / 45

Algorisme de Milner

`\f -> \(x : xs) -> f x : map f xs`

Arbre de l'expressió:

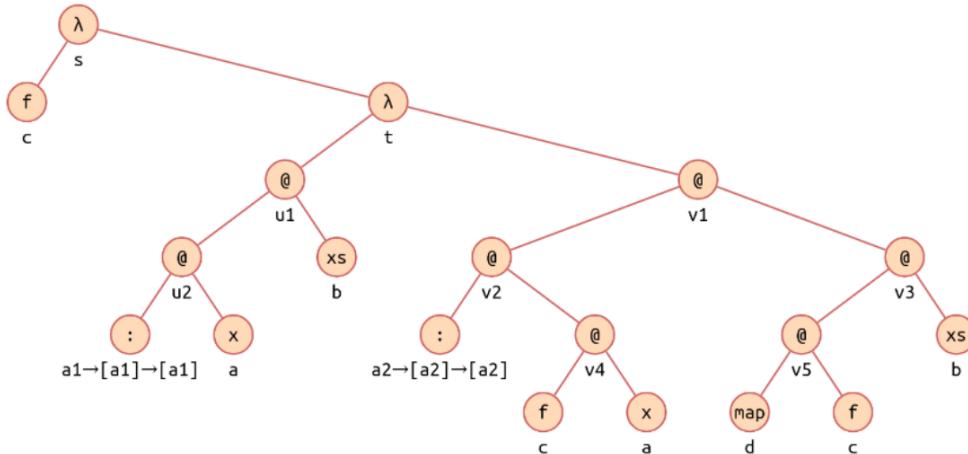


30 / 45

Algorisme de Milner

$\lambda f \rightarrow \lambda(x : xs) \rightarrow f x : \text{map } f \ xs$

Arbre etiquetat amb tipus:



31 / 45

Algorisme de Milner

$\lambda f \rightarrow \lambda(x : xs) \rightarrow f x : \text{map } f \ xs$

Equacions:

- $s = c \rightarrow t$
- $t = u_1 \rightarrow v_1$
- $u_2 = b \rightarrow u_1$
- $a_1 \rightarrow [a_1] \rightarrow [a_1] = a \rightarrow u_2$
- $v_2 = v_3 \rightarrow v_1$
- $a_2 \rightarrow [a_2] \rightarrow [a_2] = v_4 \rightarrow v_2$
- $c = a \rightarrow v_4$
- $v_5 = b \rightarrow v_3$
- $d = c \rightarrow v_5$
- $s = d$

32 / 45

Algorisme de Milner

```
\f -> \(x : xs) -> f x : map f xs
```

Solució:

- a1 = a
- b = [a]
- c = a → a2
- d = (a → a2) → [a] → [a2]
- s = (a → a2) → [a] → [a2]
- t = [a] → [a2]
- u1 = [a]
- u2 = [a] → [a]
- v1 = [a2]
- v2 = [a2] → [a2]
- v3 = [a2]
- v4 = a2
- v5 = [a] → [a2]

Per tant, el tipus de l'arrel és $s = (a \rightarrow a2) \rightarrow [a] \rightarrow [a2]$.

33 / 45

Funcions amb més d'una definició

```
map f [] = []
map f (x : xs) = f x : map f xs
```

Quan hi ha més d'una definició, apareix un bosc d'arbres.

Les definicions per la mateixa funció tenen el mateix tipus a l'arrel.

Analitzant una sola sola definició, el tipus pot ser més general que l'esperat:

```
foldr f z (x : xs) = f x (foldr f z xs)
```

```
foldr :: (t1 -> t2 -> t2) -> t3 -> [t1] -> t2 !
```

```
foldr f z (x : xs) = f x (foldr f z xs)
foldr f z [] = z
```

```
foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2 ⚡
```

34 / 45

Altres construccions

- Els `let` o `where` més simples es poden expressar amb abstraccions i aplicacions:

Per exemple

`let x = y in z`

es tracta com

$(\lambda x . z) y$

- Les guardes es tracten com un `if-then-else`.
- El `case` es tracta com una definició per patrons.

35 / 45

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

36 / 45

Classes

La presència de definicions com ara

```
(+) :: Num a => a -> a -> a  
(>) :: Ord a => a -> a -> Bool
```

introduceix unes noves *restriccions de context*.

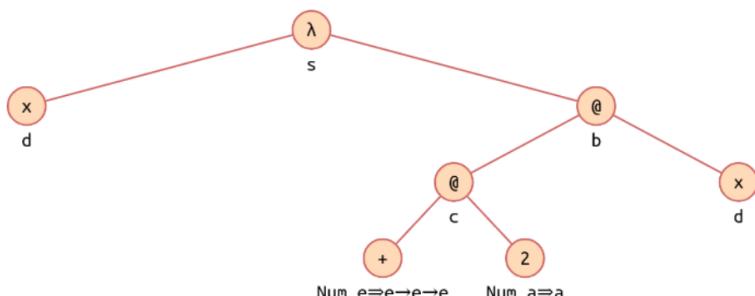
Per tant, les solucions també han de contenir i satisfer les condicions de classe.

37 / 45

Classes

f x = **2** + x

Arbre etiquetat:



38 / 45

Classes

Equacions:

- $s = d \rightarrow b$
- $c = d \rightarrow b$
- $e \rightarrow e \rightarrow e = a \rightarrow c$

Restriccions:

- Num a
- Num e

Solució:

- $s = a \rightarrow a$
- $b = a$
- $c = a \rightarrow a$
- $d = a$
- $e = a$

El tipus de l'arrel (de f) és doncs Num a $\Rightarrow a \rightarrow a$.

39 / 45

Contingut

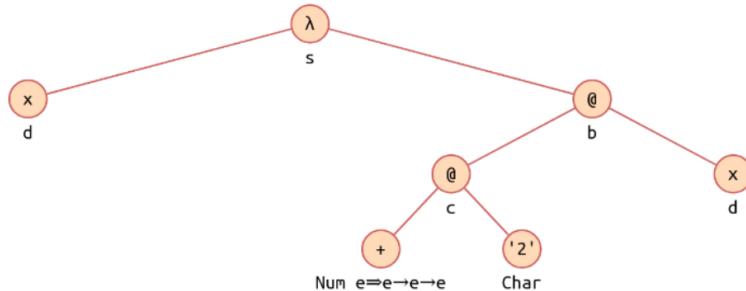
- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

40 / 45

Errors

f x = '2' + x

Arbre etiquetat:



41 / 45

Errors

Equacions:

- $s = d \rightarrow b$
- $c = d \rightarrow b$
- $e \rightarrow e \rightarrow e = \text{Char} \rightarrow c$

Restriccions:

- Num e

Intent de solució:

- $s = \text{Char} \rightarrow \text{Char}$
- $b = \text{Char}$
- $c = \text{Char} \rightarrow \text{Char}$
- $d = \text{Char}$
- $e = \text{Char}$
- Num Char **X**

Perquè Char no és instància de Num!

42 / 45

Contingut

- Introducció
- Algorisme de Milner
- Exercicis
- Funcions
- Classes
- Errors
- Exercicis

43 / 45

Exercicis

- Inferiu el tipus de:

`ones = 1 : ones`

- Inferiu el tipus de:

`even x = if rem x 2 == 0 then True else False`

amb `rem :: Int → Int → Int.`

- Inferiu el tipus de:

`even x = rem x 2 == 0`

- Inferiu el tipus de:

`last [x] = x`

Recordeu que `[x]` és `x:[]`.

44 / 45

Exercicis

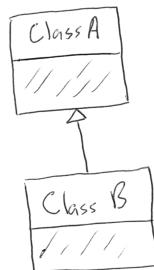
- Inferiu el tipus de:

```
delete x (y:ys) =
  if x == y
  then ys
  else y : delete x ys
```

```
amb (==) :: Eq a => a → a → Bool.
```

Llenguatges de Programació

Conceptes avançats



Jordi Petit, Fernando Orejas, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona



1 / 82

Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Concurrència

2 / 82

Recursivitat

Tail Recursion: la crida recursiva es fa just abans de retornar el valor.

Factorial recursiu:

```
def f_rec(n):
    if n == 0:
        return 1
    else:
        return n * f_rec(n-1)
```

Factorial tail recursion:

```
def f_tailrec(n, resultat = 1):
    if n == 0:
        return resultat
    else:
        return f_tailrec(n-1, n*resultat)
```

- Tail Recursion Optimization: optimització en que el compilador substitueix la crida per recursiva per un salt.
- Python no ho suporta:

```
f_rec(1000)
RecursionError: maximum recursion depth exceeded in comparison
```

- Altra opció és passar-la a iterativa.

3 / 82

Tail Recursion

Però encara és útil:

Recursivitat normal:

```
def slow_fib(n):
    if n < 2:
        return 1
    else:
        return slow_fib(n - 1) +
               slow_fib(n - 2)
```

```
fib(40) = 165580141
temps(s): 21.715350
```

Tail recursion:

```
def quick_fib(n, acc1=1, acc2=1):
    if n < 2:
        return acc1
    else:
        return quick_fib(n - 1,
                          acc1 + acc2,
                          acc1)
```

```
fib(40) = 165580141
temps(s): 0.000117
```

* Hem utilitzat el mòdul `pytictoc` per mostrar el temps d'execució.

Tail Recursion

Per què funciona?

```
>>> slow_fib(4)
4 ##          # paràmetre i pila
3 ###
2 ####
1 #####
0 #####
1 #####
2 ##
1 #####
0 #####
5           # resultat

>>> quick_fib(4)
4 ##
3 ###
2 ####
1 #####
5
```

* Hem utilitzat el mòdul `traceback` per mostrar la mida de la pila.

5 / 82

Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Concurrència

6 / 82

Continuation-Passing Style

És una tècnica de la programació funcional en la que es retornen les funcions a aplicar al resultat, en lloc dels valors.

Exemple:

```
expr = lambda: (1 + 2) * 3 + 4
expr() ⏪ 13
```

Fem una funció per cada operació.

```
mes2 = lambda x, cont: cont(2 + x)
per3 = lambda x, cont: cont(3 * x)
mes4 = lambda x, cont: cont(4 + x)

def expr_cps(x, cont):
    mes2(x, (lambda y:
              per3(y, (lambda z:
                        mes4(z, (lambda res:
                                  cont(res)))))))

expr_cps(1, print) ⏪ 13
```

Com funciona la pila?

```
expr ##
mes2 ###
per3 #####
mes4 ######
13
```

Recorda una mica als *thunks* del Haskell.

7 / 82

CPS amb funcions recursives

Amb una funció recursiva és més natural.

Exemple:

```
identitat = lambda x: x

def fact_cps(n, cont):
    if n == 0:
        return cont(1)
    else:
        return fact_cps(n - 1,
                         lambda value:
                             cont(n * value))

fact_cps(6, identitat) ⏪ 720
```

Pila?

```
fact_cps ##
fact_cps ###
fact_cps ####
fact_cps #####
fact_cps ######
fact_cps #######
fact_cps #######
fact_cps ########
720
```

Continuem tenint el problema de la pila:

```
fact_cps(1000, identitat)
RecursionError: maximum recursion depth exceeded while calling a Python object
```

8 / 82

Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Concurrència

9 / 82

Trampolining

És una tècnica que evita el creixement de la pila.

Funció trampolí:

```
def trampoline(f, *args):
    print('trampoline', len(traceback.extract_stack()) * '#')
    v = f(*args)
    while callable(v):
        v = v()
    return v
```

Factorial:

Afegim a la funció factorial un parell de *lambdes*:

```
def fact_cps2(n, cont):
    if n == 0:
        return cont(1)
    else:
        return lambda: fact_cps2(
            n - 1,
            lambda value: lambda: cont(n * value))
```

10 / 82

Execució

La pila no creix:

```
trampoline(fact_cps2, 6, identitat)
👉
trampoline ##
fact_cps2 ###
fact_cps2 ####
fact_cps2 #####
fact_cps2 #####
fact_cps2 #####
fact_cps2 #####
fact_cps2 #####
fact_cps2 #####
720
```

Funciona!

```
trampoline(fact_cps2, 1000, identitat)
👉 4023872600770.....0000000000000000
```

11 / 82

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Concurrència

12 / 82

Programació orientada a objectes

Elements principals de la POO:

- Reutilització de codi
- Modularitat
- Facilitat de manteniment
- Ampliació de funcionalitats
- Abstracció
- Encapsulació
- Herència

13 / 82

Herència i subclasses

L'herència i la relació de subclasses tenen per objectiu:

- Estructurar millor el codi.
- Reaprofitar millor el codi.
- Simplificar el disseny.

14 / 82

Herència i subclasses

Exemple:

```
class Empleat {...}

function sou(e: Empleat): number {...}

e = new Empleat()
s = sou(e)
```

Amb programació "clàssica":

```
function sou(e: Empleat): number {
    if (e.es_venedor()) {
        ...
    } else if (e.es_contable()) {
        ...
    } else if (e.es_executiu()) {
        ...
    }
}
```

Amb POO:

```
class Empleat {
    function sou(): number {...}
    ...
}

class Venedor extends Empleat {
    function sou(): number {...}
    ...
}

class Comptable extends Empleat {
    function sou(): number {...}
    ...
}
```

15 / 82

Herència i subclasses

A cada classe es poden re definir operacions de la classe base.

```
class Empleat {
    function sou(): number {...}
}

class Venedor extends Empleat {
    function sou(): number {...}
}

class Comptable extends Empleat {
    function sou(): number {...}
}
```

16 / 82

Herència i subclasses

A cada classe es poden definir noves operacions.

```
class Empleat {
    function sou(): number {...}
}

class Venedor extends Empleat {
    function comissio(): number {...}
}

class Comptable extends Empleat {
    function fulls_de_calcul(): FullCalcul[] {...}
}
```

17 / 82

Herència i subclasses

L'operació que es crida depèn de la (sub)classe de l'objecte en temps d'execució (*late binding*).

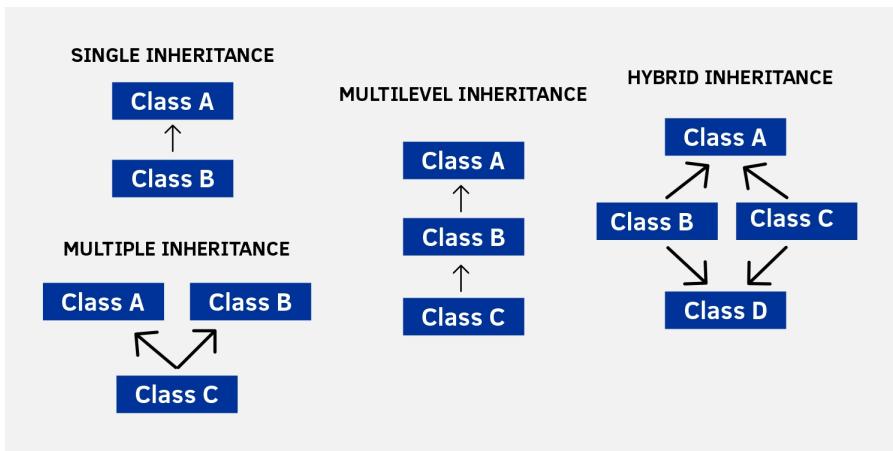
```
function escriure(e: Empleat) {
    print(e.nom, e.sou())
}

Empleat e = new Empleat()
Empleat v = new Venedor()
Empleat c = new Comptable()

escriure(e)          // usa el sou() d'Empleat
escriure(v)          // usa el sou() de Venedor
escriure(c)          // usa el sou() de Comptable
```

18 / 82

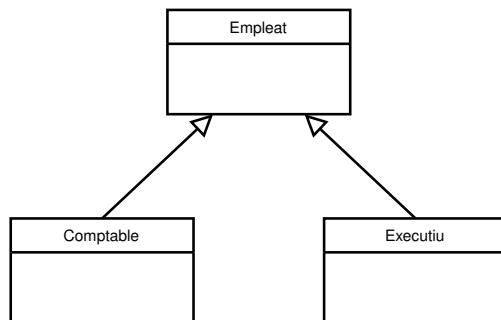
Herència



19 / 82

Herència simple

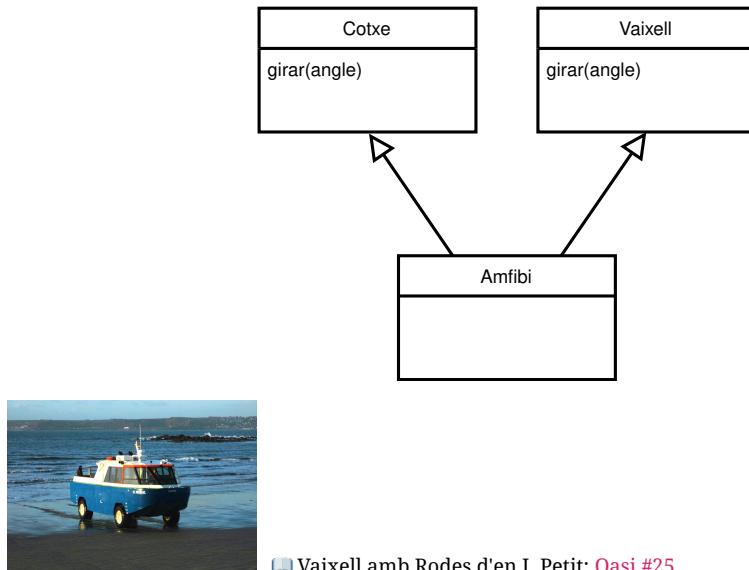
Una classe només pot ser subclasse d'una altra classe.



20 / 82

Herència múltiple

Una classe pot ser subclasse de més d'una classe.



■ Vaixell amb Rodes d'en J. Petit: Oasi #25

21 / 82

Promesa de l'00

Si es canvia l'estructura salarial:

- En programació "clàssica" cal refer del tot la funció `sou()` (i potser més operacions).
- En programació "OO", es canvien les classes i el mètode `sou()` d'algunes.

22 / 82

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Concurrència

23 / 82

Declaració de subclasses en C++

```
class Empleat { ... };
class Venedor: Empleat { ... };
```

O també:

```
class Venedor: public Empleat { ... };
class Venedor: protected Empleat { ... };
class Venedor: private Empleat { ... };
```

Amb herència múltiple:

```
class Cotxe { ... };
class Vaixell { ... };
class Hibrid: public Cotxe, public Vaixell { ... };
```

Resolució de conflictes:

```
hibrid.Cotxe::girar(90);
hibrid.Vaixell::girar(90);
```

24 / 82

Declaració de subclasses en Java

```
class Empleat { ... }  
class Venedor extends Empleat { ... }
```

En Java no hi herència múltiple amb classes, però sí amb interfícies:

```
interface Cotxe { ... }  
interface Vaixell { ... }  
class Hibrid implements Cotxe, Vaixell { ... }
```

Les interfícies de Java són com les classes de Haskell (quin embolic!).

25 / 82

Declaració de subclasses en Python

```
class Empleat:  
    ...  
class Venedor(Empleat):  
    ...
```

Amb herència múltiple:

```
class Hibrid(Cotxe, Vaixell):  
    ...
```

Resolució de conflictes:

- Quan a les dues classes hi ha mètodes amb el mateix nom, s'hereta el de la primera.

26 / 82

Visibilitat dels membres

Els LPs limiten la visibilitat dels membres (atributs i mètodes) de les classes:

Ajuda a:

- Encapsular els objectes en POO.
- Definir una interfície clara i independent de la implementació.
- Prevenir errors en el codi.

27 / 82

Visibilitat en C++

Els especificadors d'accés defineixen la visibilitat dels membres d'una classe.

```
class Classe {  
    public:  
        ...  
    protected:  
        ...  
    private:  
        ...  
};
```

- **public**: els membres són visibles des de fora de la classe
- **privat**: no es pot accedir (ni veure) als membres des de fora de la classe
- **protegit**: no es pot accedir als membres des de fora de la classe, però s'hi pot accedir des de classes heretades.

```
class Classe {  
    // privat per defecte  
};
```

```
struct Estructura {  
    // public per defecte  
};
```

28 / 82

Visibilitat en C++

Els especificadors d'accés també defineixen la visibilitat dels membres quan es deriva una classe:

```
class SubClasse: public Classe { ... };
```

- Els membres protegits de Classe són membres protegits de SubClasse.
- Els membres públics de Classe són membres públics de SubClasse.

```
class SubClasse: protected Classe { ... };
```

- Els membres protegits i públics de Classe són membres protegits de SubClasse.

```
class SubClasse: private Classe { ... };  
class SubClasse: Classe { ... };           // 'private' per defecte en classes
```

- Els membres públics i protegits de Classe són membres privats de SubClasse.

29 / 82

Visibilitat en C++

```
class A {  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};  
  
class B : public A {  
    // x és public  
    // y és protegit  
    // z no és visible des de B  
};  
  
class C : protected A {  
    // x és protegit  
    // y és protegit  
    // z no és visible des de C  
};  
  
class D : private A {  
    // x és privat  
    // y és privat  
    // z no és visible des de D  
};
```

30 / 82

Visibilitat en Java

Els nivells d'accés defineixen la visibilitat dels membres (atributs i mètodes) d'una classe.

```
class Classe {  
    public ...  
    protected ...  
    private ...  
    ...  
}
```

- **public:** aquest membre és accessible des de fora de la classe
- **privat:** no es pot accedir (ni veure) en aquest membre des de fora de la classe
- **protegit:** no es pot accedir en aquest membre des de fora de la classe, però s'hi pot accedir des de classes heretades.
- **res:** només el codi en el package actual pot accedir aquest membre.

31 / 82

Visibilitat en Java

En Java no es pot limitar la visibilitat heretant classes (sempre és "public").

```
class SubClasse extends Classe { ... }
```

- Els membres protegits de **Classe** són membres protegits de **SubClasse**.
- Els membres públics de **Classe** són membres públics de **SubClasse**.

32 / 82

Visibilitat en Java

```
package p;  
  
public class A {  
    public int a;  
    protected int b;  
    private int c;  
    int d;  
}  
  
class B extends A {  
    // a és visible des de B  
    // b és visible des de B  
    // c no és visible des de B  
    // d és visible des de B  
}  
  
// A.a és visible des de p  
// A.b és visible des de p  
// A.c no és visible des de p  
// A.d és visible des de p  
  
package q;  
  
import p.*;  
  
class C extends p.A {  
    // a és visible des de C  
    // b és visible des de C  
    // c no és visible des de C  
    // d no és visible des de C  
}  
  
// A.a és visible des de q  
// A.b no és visible des de q  
// A.c no és visible des de q  
// A.d no és visible des de q
```

33 / 82

Visibilitat en Python

En Python no hi ha restriccions de visibilitat.

Tot és visible.

Per *convenció*, els membres que comencen per `_` (però no per `__`) són privats.

34 / 82

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Concurrència

35 / 82

Tipatge estàtic i tipatge dinàmic

Tipatge estàtic: La verificació de tipus que es realitza durant la compilació del codi.

- El compilador comprova si les variables s'utilitzen de manera coherent amb el seu tipus durant la compilació del codi.
- Si hi ha un error de tipus, el compilador no genera codi.
- Ajuda a detectar i corregir errors abans d'executar el codi, evitant problemes durant l'execució.

Tipatge dinàmic: La verificació de tipus que es realitza durant l'execució del codi.

- El tipus de la variable es determina en temps d'execució
- Si hi ha un error de tipus, aquest no es detectarà fins que el codi s'executi.

36 / 82

Late binding (vinculació)

El late binding és el procés pel qual es determina (en temps d'execució) quin mètode cal cridar en funció del tipus dinàmic d'un objecte.

```
class Animal {
    parlar() {
        print("grr")
    }
}

class Gat extends Animal {
    parlar() {
        print("mèu")
    }
}

class Gos extends Animal {
    parlar() {
        print("bub")
    }
}

function parlarN(animal: Animal,
                 n: number) {
    repeat (n) {
        animal.parlar() ← late binding
    }
}
```

animal: Animal = new Animal()	
gat: Gat = new Gat()	👉 grr
gos: Gos = new Gos()	👉 mèu
animal.parlar()	👉 bub
gat.parlar()	👉 grr
gos.parlar()	👉 mèu
parlarN(animal, 3)	👉 bub bub bub
parlarN(gat, 3)	👉 grr grr grr
parlarN(gos, 3)	👉 mèu mèu mèu

37 / 82

Vinculació en Java

En Java, els objectes tenen un tipus estàtic i un tipus dinàmic:

```
Animal animal;
animal = new Gat();
```

- El tipus estàtic d'`animal` és `Animal`.
- El tipus dinàmic d'`animal` és `Gat`.

El tipus dinàmic ha de ser un subtípus del tipus estàtic.

En temps de compilació, es comprova que les crides es puguin aplicar al tipus estàtic.

En temps d'execució, la vinculació es fa en funció del tipus dinàmic.

38 / 82

Vinculació en Java

Donada una declaració `C c;` i una operació `c.m():`

- En temps de compilació, es verifica que la classe C tingui el mètode m (directament o a través d'herència).
- En temps d'execució, es crida al m de la classe corresponent al tipus dinàmic de c o de la seva superclasse més propera que l'implementi.

39 / 82

Vinculació en Java

```
class Animal {  
    void parlar() {  
        print("grr");  
    }  
  
    class Gat extends Animal {  
        void parlar() {  
            print("mèu");  
        }  
        void filar() {  
            print("rum-rum");  
        }  
    }  
  
    void parlarN(Animal animal, int n) {  
        for (int i = 0; i < n; ++i) {  
            animal.parlar();  
        }  
    }  
}  
  
Animal animal = new Animal();  
Gat gat = new Gat();  
  
animal.parlar();      ➡ grr  
gat.parlar();      ➡ mèu  
  
parlarN(animal, 3);  ➡ grr grr grr  
parlarN(gat, 3);   ➡ mèu mèu mèu  
  
gat.filar();       ➡ rum-rum  
animal.filar()     ✗ error compilació
```

40 / 82

Vinculació en Python

En Python, el tipus dels objectes és dinàmic.

```
>>> e = Empleat()
>>> v = Venedor()
>>> type(e)
<class '__main__.Empleat'>
>>> type(v)
<class '__main__.Venedor'>
>>> v = e
>>> type(v)
<class '__main__.Empleat'>
```

Donada una operació `c.m()`:

- En temps d'execució, es crida al `m` de la classe corresponent al tipus dinàmic de `c` o de la seva superclasse més propera que l'implementi.

41 / 82

Vinculació en Python

```
class Animal:
    def parlar(self):
        print("grr")

class Gat(Animal):
    def parlar(self):
        print("mèu")
    def filar(self):
        print("rum-rum")

def parlarN(animal, n):
    n * [animal.parlar()]

animal = Animal()
gat = Gat()

animal.parlar()      grr
gat.parlar()      mèu
parlarN(animal, 3)  grr grr grr
parlarN(gat, 3)  mèu mèu mèu
gat.filar();   rum-rum
animal.filar()  X error execució
```

42 / 82

Vinculació en C++

En C++, els objectes estàtics tenen un tipus estàtic.

```
Animal a = Gat();
```

- El tipus estàtic d'a és `Animal`: quan se li assigna un `Gat` es perd la part extra.
- (Recordeu: El pas per còpia fa una assignació)

Els objectes dinàmics (punters i referències) tenen un tipus estàtic i un tipus dinàmic.

```
Animal* a = new Gat();
```

- El tipus estàtic d'a és punter a `Animal`.
- El tipus dinàmic d'a és punter a `Gat`.

```
Animal& a = Gat();
```

- El tipus estàtic d'a és referència a `Animal`.
- El tipus dinàmic d'a és referència a `Gat`.

43 / 82

Vinculació en C++

Per a objectes estàtics, la vinculació és estàtica.

El tipus dinàmic ha de ser un subtípus del tipus estàtic.

En temps de compilació, es comprova que les crides es puguin aplicar al tipus estàtic.

En temps d'execució, la vinculació es fa en funció del tipus dinàmic, sobre els mètodes marcats `virtual`.

44 / 82

Vinculació en C++

```
class Animal {
    virtual void parlar() {
        print("grr");
    }
}

class Gat: Animal {
    virtual void parlar() {
        print("mèu");
    }
    virtual void filar() {
        print("rum-rum");
    }
}

void parlarN(Animal animal, n: int) {
    for (int i = 0; i <n; ++i) {
        animal.parlar();
    }
}
```

Animal animal;
Gat gat;

animal.parlar(); grr
gat.parlar(); mèu

parlarN(animal, 3); grr grr grr
parlarN(gat, 3); grr grr grr *

gat.filar(); rum-rum
animal.filar() error compilació

* Com que parlarN rep un Animal per còpia, al cridar parlarN(gat, 3) es perd la part de gat.

45 / 82

Vinculació en C++

```
class Animal {
    virtual void parlar() {
        print("grr");
    }
}

class Gat: Animal {
    virtual void parlar() {
        print("mèu");
    }
    virtual void filar() {
        print("rum-rum");
    }
}

void parlarN(Animal* animal, n: int) {
    for (int i = 0; i <n; ++i) {
        animal->parlar();
    }
}
```

Animal animal;
Gat gat;

animal.parlar(); grr
gat.parlar(); mèu

parlarN(animal, 3); grr grr grr
parlarN(&gat, 3); mèu mèu mèu *

gat.filar(); rum-rum
animal.filar() error compilació

* Com que parlarN rep un punter a Animal, al cridar parlarN(gat, 3) el tipus dinàmic continua sent Gat.

46 / 82

Vinculació en C++

```
class Animal {
    virtual void parlar() {
        print("grr");
    }
}

class Gat: Animal {
    virtual void parlar() {
        print("mèu");
    }
    virtual void filar() {
        print("rum-rum");
    }
}

void parlarN(Animal& animal, n: int) {
    for (int i = 0; i <n; ++i) {
        animal.parlar();
    }
}
```

```
Animal animal;
Gat gat;

animal.parlar();      ↗ grr
gat.parlar();        ↗ mèu

parlarN(animal, 3); ↗ grr grr grr
parlarN(gat, 3);   ↗ mèu mèu mèu *
```

gat.filar(); ↗ rum-rum
animal.filar() ✗ error compilació

* Com que parlarN rep un Animal per referència, al cridar parlarN(gat, 3) el tipus dinàmic continua sent Gat.

47 / 82

Vinculació en C++

```
class Animal {
    void parlar() {
        print("grr");
    }
}

class Gat: Animal {
    void parlar() {
        print("mèu");
    }
}

void parlarN(Animal& animal, n: int) {
    for (int i = 0; i <n; ++i) {
        animal.parlar();
    }
}
```

```
Animal animal;
Gat gat;

parlarN(animal, 3); ↗ grr grr grr
parlarN(gat, 3);  ↗ grr grr grr *
```

* Com que parlar no és virtual, parlarN no fa late binding.

48 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Concurrència

49 / 82

Noció de subtipus

Definició 1:

`s` és subtipus de `t` si tots els valors d'`s` són valors de `t`.

Exemple en Pearl:

```
subset Evens of Int where {$_ % 2 == 0}
```

► Aquesta mena de subtipus no són habituals en els LPs.

50 / 82

Noció de subtipus

Definició 2:

s és subtipus de t si qualsevol funció que es pot aplicar a un objecte de tipus t es pot aplicar a un objecte de tipus s.

Exemple en C++:

```
class Forma;
class Quadrat: Forma;           // Forma és subtipus de Forma

double area(const Forma& f);

Forma f;
area(f);           ✓
Quadrat q;
area(q);           ✓
```

► Aquesta és la definició en què es basa la programació orientada a objectes.

51 / 82

Noció de subtipus

Definició 2':

s és subtipus de t si en tot context que es pot usar un objecte de tipus t es pot usar un objecte de tipus s.

► Aquesta és la definició en què (a vegades es diu que) es basa la programació orientada a objectes.

52 / 82

Noció de subtipus

Les definicions 1 i 2 no són equivalents:

- Si s és subtipus de t segons la Def. 1,
llavors també ho és d'acord amb la Def. 2.
- La inversa, en general, no és certa. És a dir, si s és subtipus de t d'acord
amb la Def. 2, llavors no té perquè ser-ho d'acord amb la Def. 1.

Exemple:

```
class T {  
    int x;  
};  
  
class S : T {  
    int y;  
}
```

Els valors de S no es poden veure com un subconjunt dels valors de T , ja
que tenen més elements.

53 / 82

Noció de subtipus

Definició 3:

s és subtipus de t si tots els objectes de s es poden convertir implícitament a
objectes de t (*type casting* o coerció).

54 / 82

Comprovació i inferència amb subtipus

- Si $e :: s \leq t$, llavors $e :: t$.
- Si $e :: s, s \leq t \text{ if } f :: t \rightarrow t'$, llavors $f e :: t'$.

La notació $e :: t$ indica que e és de tipus de t .

La notació $s \leq t$ indica que s és un subtipus de t .

55 / 82

Comprovació i inferència amb subtipus

- Si $e :: s \leq t$, llavors $e :: t$.
- Si $e :: s, s \leq t \text{ if } f :: t \rightarrow t'$, llavors $f e :: t'$.

Per tant,

- Si $e :: s, s \leq t \text{ if } f :: t \rightarrow t$, llavors $f e :: t$.

Però no podem assegurar que $f e :: s$! Per exemple, si tenim

- $x :: \text{parell}$
- $\text{parell} \leq \text{int}$
- $\text{function es_positiu(int): boolean}$
- $\text{function incrementa(int): int}$

Llavors

- $\text{es_positiu}(x) :: \text{bool} \checkmark$
- $\text{incrementa}(x) :: \text{int} \checkmark$
- $\text{incrementa}(x) :: \text{parell} \times$

56 / 82

El cas de l'assignació

- Si $x :: t \in e :: s \leq t$, llavors $x = e$ és una assignació correcta.
- Si $x :: s \in e :: t \leq s$, llavors $x = e$ és una assignació incorrecta.

Exemples:

- Si $x :: \text{int} \in e :: \text{parell}$, $x = e$ no té problema.
- Si $x :: \text{parell} \in e :: \text{int}$, $x = e$ crearia un problema: e potser no és parell.

57 / 82

El cas de les funcions

- Si $s \leq t \leq s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

58 / 82

El cas de les funcions

- Si $s \leq t$ i $s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

No!

Suposem que $f :: \text{parell} \rightarrow \text{parell}$ i que $g :: \text{int} \rightarrow \text{int}$.

Si $(s \rightarrow s') \leq (t \rightarrow t')$, llavors sempre que puguem usar g , podem usar f al seu lloc. Com que $g\ 5$ és legal, $f\ 5$ també seria legal. Però f espera un parelli 5 no ho és.

59 / 82

El cas de les funcions

- Si $s \leq t$ i $s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

No!

Suposem que $f :: \text{parell} \rightarrow \text{parell}$ i que $g :: \text{int} \rightarrow \text{int}$.

Si $(s \rightarrow s') \leq (t \rightarrow t')$, llavors sempre que puguem usar g , podem usar f al seu lloc. Com que $g\ 5$ és legal, $f\ 5$ també seria legal. Però f espera un parelli 5 no ho és.

- En canvi, si $s \leq t$ i $s' \leq t'$, llavors $(t \rightarrow s') \leq (s \rightarrow t')$ és correcte.

60 / 82

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } s \leq \text{List } t$?

61 / 82

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } s \leq \text{List } t$?

No!

```
class Animal
class Gos extends Animal
class Gat extends Animal

function f(animals: List<Animal>) {
    animals.push(new Gat())           // perquè no?
}

gossos: List<Gos> = ...
f(gossos)                         // ai, ai
```

62 / 82

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } t \leq \text{List } s$?

63 / 82

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } t \leq \text{List } s$?

No!

```
class Animal
class Gos extends Animal {
    function borda() {...}
}
class Gat extends Animal;

function f(gossos: List<Gos>) {
    for (var gos: Gos of gossos) gos.borda()
}

List<Animal> animals = [new Gos(), new Animal, new Gat()]
f(animals) // alguns animals no borden 🐕
```

64 / 82

Variància de constructors de tipus

Sigui c un constructor de tipus i sigui $s \leq t$.

- Si $c\ s \leq c\ t$, llavors c és covariant.
- Si $c\ t \leq c\ s$, llavors c és contravariant.
- Si no és covariant ni contravariant, llavors c és invariant.

65 / 82

Variància de constructors de tipus

Sigui c un constructor de tipus i sigui $s \leq t$.

- Si $c\ s \leq c\ t$, llavors c és covariant.
- Si $c\ t \leq c\ s$, llavors c és contravariant.
- Si no és covariant ni contravariant, llavors c és invariant.

Hem vist doncs que:

- El constructor `->` és contravariant amb el primer paràmetre.
- El constructor `->` és covariant amb el segon paràmetre.
- El constructor `List` és invariant.

66 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- **Clausures**
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Concurrència

67 / 82

Clausures

Al retornar funcions es creen clausures (*closure*):

- tanca l'abast (*scope*) lèxic del voltant i capture els seus valors.

```
def interna(x):
    z = "!"
    return lambda y: print(x, y, z)

externa = interna("Hola")

externa("món") ➔ Hola món !
```

Fixeu-vos en que si la funció interna tornés només una funció, no funcionaria.

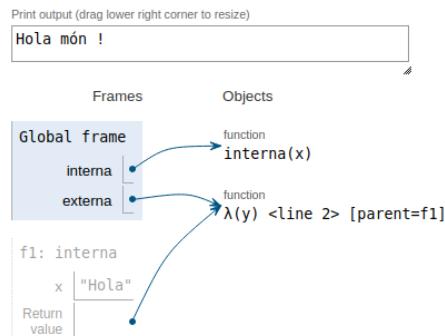


diagrama: [Python Tutor](#)

68 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- **Clausures**
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Concurrència

69 / 82

Partial

Podem currificar en python?

```
def partial ( f , x ):  
    def g(* args ):  
        return f (*(( x ,) + args ))  
    return g
```

Exemple:

```
multiplica = lambda x, y: x * y  
doble = partial(multiplica , 2)  
  
doble(3) ➔ 6
```

70 / 82

Partial++

Més general:

```
def partialN (* args ):
    def g(* args2 ):
        f = args [0]
        xs = args [1:] + args2
        return f (* xs)
    return g
```

Exemple:

```
from functools import reduce

def sumaRed(*args):
    suma = lambda x, y: x + y
    return reduce(suma, list ( args ), 0)

sumaRed(1,2,3,4) ➔ 10

sumaRed10 = partialN(sumaRed, 1, 2, 3, 4)
print(sumaRed10(5, 5, 5)) ➔ 25
```

71 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- **Clausures**
 - *Partial*
 - **Objectes sense classe**
 - Memorització
 - Decoradors
- Concurrència

72 / 82

Objecte punt sense classe

```
from math import pi, atan2, degrees

def punt(x, y):
    def temp(*args):
        if args[0] == 'crt':
            return x, y
        elif args[0] == 'plr':
            return (x ** 2 + y ** 2) ** 0.5, \
                   degrees(atan2(y, x))
        elif args[0] == 'dst':
            p2 = args[1]('crt')
            return ((x-p2[0])**2 + (y-p2[1])**2)**0.5
    return temp

punt(1, 0)('crt') ➔ (1, 0)
punt(1, 0)('plr') ➔ (1.0, 0.0)
punt(1, 1)('crt') ➔ (1, 1)
punt(1,1)('plr') ➔ (1.4142135623730951, 45.0)
punt(1, 1)('dst', punt(2, 0)) ➔ 1.4142135623730951
punt(1, 1)('dst', punt(1, 1)) ➔ 0.0
```

73 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- **Clausures**
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Concurrència

74 / 82

Test de funcions

```
from pytictoc import TicToc

def test(n):
    def prec(x):
        return '{:.6f}'.format(x)

    def clausura(f):
        t = TicToc()
        t.tic()
        print('f(', n, ') = ', f(n), sep=' ')
        print('temps(s):', prec(t.tocvalue()))

    return clausura

def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)

test40 = test(40)
test40(fib)
f(40) = 102334155
temps(s): 23.586690
```

75 / 82

Memorització genèrica

```
def memoriza (f):
    mem = [] # la memòria

    def f2 (x):
        if x not in mem:
            mem.append(x)
            mem[-1] = f(x)
        return mem[-1]

    return f2

fib = memoriza(fib)
test40(fib)
f(40) = 102334155
temps(s): 0.000051
```

76 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- **Clausures**
 - *Partial*
 - Objectes sense classe
 - Memorització
 - **Decoradors**
- Concurrència

77 / 82

Decoradors

Són un mètode per alterar quelcom invocable (*callable*).

Ho podem fer mitjançant les clausures.

```
def testDec(f):
    def wrapper(*args):
        valor = f(*args)
        print('fib(' + str(args[0]) + ') = ' + \
              str(valor))
        return valor

    return wrapper

@testDec
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)
```

```
test4 = test(4)
test4(fib)
fib(1) = 1
fib(0) = 0
fib(2) = 1
fib(1) = 1
fib(3) = 2
fib(1) = 1
fib(0) = 0
fib(2) = 1
fib(4) = 3
f(4) = 3
temps(s): 0.000089
```

Funciona també aplicant `fib = memoritza(fib)`.

78 / 82

Decoradors parametritzats

Podem afegir arguments parametritzant els decoradors:

```
def testInterval(inici, fi):
    def decorador(f):
        def wrapper(*args):
            valor = f(*args)
            n = args[0]
            if inici <= n <= fi:
                print('fib(' + str(n) + ') = ' + \
                      str(valor))
            return valor
        return wrapper
    return decorador

@testInterval(35, 40)
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)

test40(fib)
fib(35) = 9227465
fib(36) = 14930352
fib(37) = 24157817
fib(38) = 39088169
fib(39) = 63245986
fib(40) = 102334155
f(40) = 102334155
temps(s): 0.000119
```

79 / 82

Memorització genèrica amb decoradors

```
def memoriza (f):
    mem = []
    def f2 (x):
        if x not in mem:
            mem[x] = f(x)
        return mem[x]
    return f2

@testInterval(38, 40)
@memoriza
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)

fib(38) = 39088169
fib(39) = 63245986
fib(38) = 39088169
fib(40) = 102334155
f(40) = 102334155
temps(s): 0.000078
```

80 / 82

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Concurrència

81 / 82

Concurrència

- teoria
- Async + CPS
- async i telegram

82 / 82

Llenguatges de Programació

Conceptes bàsics



Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

FIB

1 / 12

Treball dirigit

El treball dirigit consisteix en preparar un vídeo i un document escrit sobre les capacitats i límits de fer programació funcional en un llenguatge de programació imperatiu

L'objectiu és satisfer les competències transversals de l'assignatura:

Comunicació eficaç oral i escrita G4 3

Comunicar-se de manera clara i eficient en presentacions orals i escriptes sobre temes complexos, adaptant-se a la situació, al tipus de públic i als objectius de la comunicació, utilitzant les estratègies i els mitjans adequats. Analitzar, valorar i respondre adequadament a les preguntes de l'auditori.

Ús solvent dels recursos d'informació G6.3

Planificar i utilitzar la informació necessària per a un treball acadèmic (per exemple, per al treball de final de grau) a partir d'una reflexió crítica sobre els recursos d'informació utilitzats. Gestionar la informació de manera competent, independent i autònoma. Avaluar la informació trobada i identificar-ne les llacunes.

Treball dirigit

El treball dirigit consisteix en preparar un vídeo i un document escrit sobre els elements de programació funcional existents en un llenguatge de programació imperatiu. Heu de presentar, de forma informada, raonada i crítica, els elements de programació funcionals disponibles en el llenguatge, tot exposant les seves capacitats i límits.

L'avaluació es farà per avaluació entre companys (co-avaluació) utilitzant [Mussol](#).

Per identificar l'LP imperatiu del vostre treball, calculeu el vostre DNI mòdul 8 per a obtenir un número entre 0 i 7. Aquest número us indicarà l'LP que us toca:

- 0: Java
- 1: C#
- 2: Python
- 3: JavaScript
- 4: TypeScript
- 5: Rust
- 6: Julia
- 7: Scala

3 / 12

Etapes del Treball Dirigit

1. Cercar i consultar recursos d'informació sobre el tema que us ha tocat.
2. Preparar el vídeo i el document.
3. Lliurar el vídeo i el document PDF abans del 22 de maig a les 8:00 (matí).
4. Avaluuar els vídeos i els documents de 3 altres companys usant una rúbrica preparada pels professors abans del 10 de juny a les 8:00 (matí).

Nota: Donat que aquests terminis s'han publicat amb molta antelació, no s'acceptaran lliuraments amb retard.

4 / 12

Vídeo

- Presentació d'entre 8 i 10 minuts (ni més ni menys).
- Producció molt simple (no hi perdeu el temps!):
 - Veu en off amb text i material gràfic que doni suport a l'explicació.
 - No s'ha de veure l'orador.
- Característiques tècniques:
 - Resolució de 848×480 píxels com a mínim.
 - Límit de 50MB per fitxer.
 - Important: una imatge clara i un àudio molt nítid.
- El vídeo ha de ser totalment anònim (compte amb les metadades!) (malgrat que es pugui reconèixer la veu de l'orador...).

5 / 12

Document

- Lliurament d'un document curt en PDF (8 a 12 planes aproximadament).
- Primera part: acompanyament més detallat de la presentació.
- Segona part: estudi bibliogràfic.
- El document ha de ser totalment anònim (compte amb les metadades!).

6 / 12

Comunicació eficaç oral i escrita

- Vídeo i primera part del document:
 - Doneu una breu descripció i propòsit de l'LP imperatiu que us ha tocat, utilitzant conceptes introduïts al curs com ara paradigma, sistema d'execució, sistema de tipus, etc.
 - Expliqueu els elements de programació funcional disponibles en aquest LP. Com a exemple, mireu els [elements de programació funcional que hi ha en C++](#).
 - Exposeu de forma informada, raonada i crítica les seves capacitats i limitacions d'aquests elements de programació funcional dins de l'LP imperatiu.

7 / 12

Ús solvent dels recursos d'informació

- Segona part del document:
 - Descripció de les fonts d'informació emprades.
 - Useu vàries fonts.
 - Busqueu codi.
 - Avalueu la qualitat de la informació trobada.
 - Incloeu referències bibliogràfiques.
 - Millor si no totes són d'internet.

Observació: L'estudi bibliogràfic té molt de pes a la correcció.

8 / 12

Avaluació del Treball Dirigit

L'avaluació dels companys a través de la rúbrica donarà lloc a tres qualificacions:

- Nota de la competència G4.3 (A, B, C, D, NA)
- Nota de la competència G6.3 (A, B, C, D, NA)
- Nota de la part de treball dirigit de l'assignatura (0 a 10)

Casos excepcionals:

- No enviar el treball complet (vídeo i document) o no avaluar tots els treballs dels 3 companys dóna lloc a una qualificació de NA, NA, 0.
- Si algun estudiant considera que ha estat mal avaluat pels seus companys, podrà demanar que els professors li corregeixin independentment el seu treball. Les qualificacions resultants substituiran a les dels companys, ja sigui a l'alta o a la baixa.
- Els professors també podran corregir d'ofici treballs arbitraris i canviar-ne les qualificacions.
- Els professors penalitzaran als estudiants que avaluïn de forma manifestament incorrecta als seus companys.

9 / 12

Mussol

Tota la gestió dels TD es fa a utilitzant **Mussol**.

Per identificar-vos a Mussol, feu servir el vostre correu de la UPC i la vostra contrasenya del Jutge.

Podeu fer diversos enviaments a Mussol abans de la data límit. El darrer enviament és el que es tindrà en compte.

El sistema Mussol és nou, si hi trobeu cap problema, sisplau contacteu amb els professors.

10 / 12

Repetidors

Els repetidors poden reusar les seves notes del TD del curs passat.

Per a fer-ho, només cal que no enviïn el seu treball. Si hi participen, vol dir que tornen a fer tot el treball dirigit (enviament i correccions) i rebutgen la nota anterior.

11 / 12

Documentació Biblioteca UPC

Tingueu en compte aquestes pautes:

- Ús ètic de la informació
- Evitar el plagi
- Citar i elaborar la bibliografia
- Formació BIBtips

12 / 12