

# Llenguatges de Programació, FIB, 18 de gener de 2021

*L'examen dura dues hores i mitja. Es valorarà la concisió, claredat i brevetat a més de la completesa i l'exactitud de les respostes. Contesteu cada problema en un full diferent i poseu el vostre nom a cada full. Feu bona lletra. Tots els problemes puntuen igual.*

## 1. El Trivial d'LP

Contesteu les preguntes següents. Totes les respostes han de ser extremadament curtes.

1. Aparelleu cares i llenguatges de programació:



(a)

Python



(b)

COBOL



(c)

Perl



(d)

LISP



(e)

ADA



(f)

Pascal

2. Quin llenguatge de programació ha dominat en la computació científica des dels anys seixanta?
3. I quin llenguatge de programació ha dominat en les aplicacions de negocis des dels anys seixanta?
4. Quins han estat els llenguatges de programació més influencials en l'àmbit de la intel·ligència artificial?
5. Expliqueu perquè Java és o no és *type safe*.
6. Què vol dir *currifcar* una funció?
7. En quin llenguatge de programació està escrit Unix?
8. Quin llenguatge de programació fou el primer a introduir elements d'orientació a objectes?
9. Què és un *decorador* (en Python)?
10. Quin és el primer llenguatge de programació completament orientat a objectes?
11. Quin llenguatge de programació va intentar definir el Departament de Defensa dels Estats Units per utilitzar-lo en tot el seu programari?
12. Què és una *clausura* (en programació)?
13. Què és una *classe abstracta*?
14. Quin mecanisme de pas de paràmetres utilitza Java?
15. Quantes lliures esterlines costava la Màquina de Turing quan es va inventar?

## 2. Haskell

Considereu arbres binaris genèrics definits de la forma següent:

**data** *Tree* *a* = *Empty* | *Node* *a* (*Tree* *a*) (*Tree* *a*)

Primer, implementeu (recursivament) una funció d'ordre superior

*treeFold* :: (*a* → *b* → *b* → *b*) → *b* → *Tree* *a* → *b*

que generalitzi els *folds* de les llistes als arbres: L'agregat d'un node es calcula amb una funció (el primer paràmetre de *treeFold*) que combina l'arrel, l'agregat del fill esquerre i l'agregat del fill dret. L'agregat d'un arbre buit és un valor donat de tipus *b* (el segon paràmetre de *treeFold*).

A continuació, implementeu les funcions següents en termes de *treeFold* (sense usar recursivitat directament).

1. *size* :: *Tree* *a* → **Int**  
Donat un arbre binari, retorna el seu nombre de nodes.
2. *height* :: *Tree* *a* → **Int**  
Donat un arbre binari, retorna la seva alçada.
3. *treeMap* :: (*a* → *b*) → *Tree* *a* → *Tree* *b*  
Donat un arbre binari i una funció, retorna el mateix arbre, però havent aplicat la funció donada a l'element de cada node.
4. *inOrder* :: *Tree* *a* → [*a*]  
Donat un arbre binari, retorna una llista amb els elements de l'arbre en inordre.
5. *isBST* :: **Ord** *a* ⇒ *Tree* *a* → **Bool**  
Donat un arbre binari, indica si és un Binary Search Tree (arbre binari de cerca).

Pista: Totes les funcions es poden definir ben elegantment amb un parell de línies.

Finalment, feu que *Tree* sigui instància de **Functor** i de **Show** (quatre línies més). L'arbre *Node 1 (Node 2 Empty Empty) Empty* s'hauria de mostrar com a <1 <2 \* \*> \*>. No podeu usar recursivitat, però podeu usar la simpàtica i útil funció *treeFold*.

## 3. Inferència de tipus

Inferiu el tipus més general de la funció *unicorn* definida per:

*unicorn rainbow* = (**Just** *rainbow*) <\*> (**Just** (2 :: **Int**))

Per a fer-ho, dibuixeu el seu arbre de sintàxi, etiqueteu els nodes amb els seus tipus i genereu metodològicament les restriccions de tipus i classes. Expliqueu els passos realitzats.

Recordeu que l'operador <\*> és una operació de la classe *Applicative*:

**class** *Applicative* *f* **where**  
(<\*>) :: *f* (*a* → *b*) → (*f* *a* → *f* *b*)

## 4. Python

Considereu aquest fragment de programa en Python i especifiqueu què calculen les sis funcions misterioses. Per a cada funció, la vostra resposta hauria de ser el breu comentari o *docstring* que documentaria el comportament de la funció (independentment de la seva implementació).

A més, digueu i justifiqueu quin és el cost asimptòtic de la funció *mystery4(n)*.

```
def rec (n, t, f):
    if n == 0:
        return t
    else:
        return f (n , rec (n - 1, t, f ) )

def mul (a, b):
    return a * b

def add (a, b):
    return a + b

def rnd (a, b):
    x = random.random() # nombre real aleatori entre 0 i 1
    y = random.random() # nombre real aleatori entre 0 i 1
    if math.sqrt(x*x + y*y) ≤ 1:
        return b + 1
    else:
        return b

def mystery1 (n):
    return rec(n, 1, mul)

def mystery2 (n):
    return rec(n, 0, add)

def mystery3 (n):
    return rec(n, 1, lambda a, b: a + 1)

def mystery4 (n):
    return rec(n, [], lambda x, y: [x] + y)

def mystery5 (n):
    return rec(n, 0, rnd) / n * 4

def mystery6 (n):
    def f (_, xss):
        return [xs + [0] for xs in xss] + [xs + [1] for xs in xss]

    return rec(n, [[]], f)
```

## 5. Compilació

### 1. Què vol dir que una gramàtica és ambigua?

ANTLR accepta gramàtiques ambigües, però si escribim una gramàtica pel  $\lambda$ -càlcul com aquesta:

```
grammar LambdaCalculus ;

root    :   expr EOF ;

expr    :   VAR
        |   '\\\ ' VAR '.' expr
        |   '(' expr ')'
        |   expr expr
        ;

VAR     :   [a-z] ;
```

ens trobem que interpreta  $\lambda x.xy$  com a  $(\lambda x.x)y$  quan hauria de ser  $\lambda x.(xy)$  i, per tant, no funciona bé.

En canvi, si utilitzem dues produccions *expr* amb aquesta gramàtica:

```
grammar LambdaCalculus ;

root    :   expr EOF ;

expr    :   VAR
        |   '\\\ ' VAR '.' expr
        |   '(' expr ')'
        |   expr2 expr
        ;

expr2   :   VAR
        |   '(' expr ')'
        |   expr2 expr2
        ;

VAR     :   [a-z] ;
```

continuemem tenint una gramàtica ambigua però que, almenys, interpreta bé expressions  $x\lambda y.yzt$  com a  $x(\lambda y.(yz)t)$ .

### 2. Doneu una gramàtica no ambigua que interpreti bé les expressions de $\lambda$ -càlcul, utilitzant tres produccions *expr*.