

# Llenguatges de Programació, FIB, 9 de gener de 2025

*L'examen dura dues hores. Poseu el vostre nom a cada full. Contesteu cada problema enfulls diferents. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de lesrespostes. Feu bona lletra.*

## 1 $\lambda$ -càlcul (2 punts)

Recordeu que en  $\lambda$ -càlcul:

$$\begin{aligned}\mathbf{true} &\equiv \lambda t. \lambda f. t \\ \mathbf{false} &\equiv \lambda t. \lambda f. f \\ \mathbf{not} &\equiv \lambda g. g \mathbf{false} \mathbf{true} \\ \mathbf{if} &\equiv \lambda c. \lambda x. \lambda y. cxy\end{aligned}$$

1. Doneu una definició de la funció **xor** en  $\lambda$ -càlcul.
2. Comproveu que el resultat d'avaluar **xor true true** en ordre normal és **false**.

## 2 Inferència de tipus (3 punts)

Recordeu els tipus d'aquests operadors i funcions habituals:

$$\begin{aligned}\mathbf{length} :: \mathbf{Foldable}\;t &\Rightarrow t\;a \rightarrow \mathbf{Int} \\ (\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\ (\gg=) :: \mathbf{Monad}\;m &\Rightarrow m\;a \rightarrow (a \rightarrow m\;b) \rightarrow m\;b\end{aligned}$$

1. Inferiu el tipus complet de l'expressió  
 $(\mathbf{length}\;(\cdot))\;(\gg=)$
2. Un cop deduït el tipus, digueu un tipus d'objecte al que es podria aplicar aquesta expressió i doneu un exemple d'aplicació.

### 3 Avaluació mandrosa

(2 punts)

La fórmula de Taylor per calcular  $\frac{\pi}{4}$  és:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

1. Definiu una llista infinita *pi4* en Haskell amb la funció **map** per obtenir la sèrie de Taylor anterior codificant les fraccions en tuples. Per exemple, **take 8 pi4** hauria de retornar  $[(1, 1), (-1, 3), (1, 5), (-1, 7), (1, 9), (-1, 11), (1, 13), (-1, 15)]$ .
2. Considereu les definicions següents:

$$\begin{aligned}\mathbf{map}\,f\,[] &= [] && \text{-- M1} \\ \mathbf{map}\,f\,(x:xs) &= f\,x : \mathbf{map}\,f\,xs && \text{-- M2}\end{aligned}$$

$$\begin{aligned}\mathbf{take}\,0\, _ &= [] && \text{-- T1} \\ \mathbf{take}\,n\,(x:xs) &= x : \mathbf{take}\,(n-1)\,xs && \text{-- T2}\end{aligned}$$

Detalleu, pas a pas i anotant l'expressió utilitzada, com Haskell evalua l'expressió

**take 3 pi4**

3. Definiu un generador *pi4* en Python equivalent al de Haskell. Per exemple,

```
>>> g = pi4()
>>> [next(g) for _ in range(8)]
[(1, 1), (-1, 3), (1, 5), (-1, 7), (1, 9), (-1, 11), (1, 13), (-1, 15)]
```

## 4 Conceptes variats

(3 punts)

Responeu aquest exercici de forma breu.

1. Doneu en Python una implementació recursiva final (*tail recursion*) de la funció **map** per llistes.
2. Quin és el cost asimptòtic en el cas pitjor de la funció que acabeu d'implementar?
3. Què escriu aquest programa en C++? Perquè?

```
struct A {  
    virtual void f() {  
        cout << "A";  
    }  
};  
  
struct B: public A {  
    virtual void f() {  
        cout << "B";  
    }  
};  
  
void g(A* a) {  
    a->f();  
}  
  
int main() {  
    B b = B();  
    g(&b);  
    A* ap = new B();  
    g(ap);  
    A a = B();  
    g(&a);  
}
```

4. Expliqueu la noció de subtipus amb la assignació.
5. Què escriu el següent program en Python?

```
def misteri(n):  
    def misteri2(mult):  
        mult *= n;  
        return secret * mult;  
    secret = 4  
    n += 2  
    return misteri2  
  
def misteri3(param):  
    def misteri4(bonus):  
        return param(6) + bonus  
    return misteri4  
  
h = misteri(3)  
j = misteri3(h)  
print(j(2))
```

6. Descriuix breument què és una corutina.

# Llenguatges de Programació, FIB, 12 de gener de 2024

## Possible solució

### 1 $\lambda$ -càlcul

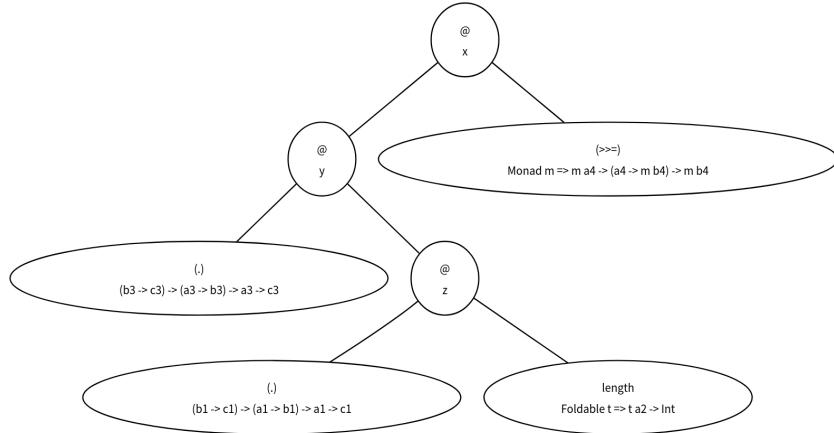
1.	$\text{xor} \equiv \lambda a. \lambda b. \text{if } a (\text{not } b) b$	
2.	$\begin{aligned} &\text{xor true true} \\ &(\lambda a. \lambda b. \text{if } a (\text{not } b) b) \text{ true true} \\ &(\lambda b. \text{if true (not } b) b) \text{ true} \\ &\text{if true (not true) true} \\ &(\lambda c. \lambda x. \lambda y. cxy) \text{ true (not true) true} \\ &(\lambda x. \lambda y. \text{true } xy) (\text{not true}) \text{ true} \\ &(\lambda y. \text{true (not true)} y) \text{ true} \\ &\text{true (not true) true} \\ &(\lambda t. \lambda f. t) (\text{not true}) \text{ true} \\ &(\lambda f. \text{not true}) \text{ true} \\ &\text{not true} \\ &(\lambda g. g \text{ false true}) \text{ true} \\ &(\lambda t. \lambda f. t) \text{ false true} \\ &(\lambda f. \text{false}) \text{ true} \\ &\text{false} \end{aligned}$	<p>definició de <b>xor</b> <math>\beta</math>-reducció de <i>a</i> <math>\beta</math>-reducció de <i>b</i> definició de <b>if</b> <math>\beta</math>-reducció de <i>c</i> <math>\beta</math>-reducció de <i>x</i> <math>\beta</math>-reducció de <i>y</i> definició de <b>true</b> <math>\beta</math>-reducció de <i>t</i> <math>\beta</math>-reducció de <i>f</i> definició de <b>not</b> <math>\beta</math>-reducció de <i>g</i> <math>\beta</math>-reducció de <i>t</i> <math>\beta</math>-reducció de <i>f</i> q.e.d.</p>

Errors freqüents:

- Ús incorrecte dels parèntesis, per l'associativitat
- Poca rigurositat en l'aplicació de les  $\beta$ -reduccions
- No indicar les operacions

## 2 Inferència de tipus

1.  $(.) \ ((.) \ \text{length}) \ (\gg=)$



(z) Foldable t

$$\begin{aligned} b1 &= t \ a2 \\ c1 &= \text{Int} \\ z &= (a1 \rightarrow b1) \rightarrow a1 \rightarrow c1 \\ &= (a1 \rightarrow t \ a2) \rightarrow a1 \rightarrow \text{Int} \end{aligned}$$

(y) b3 = a1 → t a2

$$\begin{aligned} c3 &\rightarrow a1 \rightarrow \text{Int} \\ y &= (a3 \rightarrow b3) \rightarrow a3 \rightarrow c3 \\ &= (a3 \rightarrow a1 \rightarrow t \ a2) \rightarrow a3 \rightarrow a1 \rightarrow \text{Int} \end{aligned}$$

(x) Monad m

$$\begin{aligned} m \ a4 &= a3 \\ a4 \rightarrow m \ b4 &= a1 \\ m \ b4 &= t \ a2 \\ m &= t \\ b4 &= a2 \\ x &= a3 \rightarrow a1 \rightarrow \text{Int} \\ &= m \ a4 \rightarrow (a4 \rightarrow m \ b4) \rightarrow \text{Int} \end{aligned}$$

**Solució:** Foldable m, Monad m  $\Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow \text{Int}$

2. Es podria aplicar a llistes degut al length, Monad i Foldable. Un exemple d'aplicació seria:

```
f = (length .) . (>=)
g = \x -> if even x then [x+1,x-1] else []
f [1..4] g // resultat: 4
```

**Errors freqüents:**

- No fer bé l'arbre per no ficar  $(.)$  en prefixe
- Errors en la unificació per l'associativitat a la dreta

### 3 Avaluació mandrosa

```

1. pi4 :: [(Integer, Integer)]
   pi4 = (1,1):map f pi4
      where
         f :: (Integer, Integer) -> (Integer, Integer)
         f (x, y) = (-x, y+2)

2. take 3 pi4
   = take 3 ((1,1) : map f pi4)                                -- pi4
   = (1,1) : take (3-1) (map f pi4)                            -- T2
   = (1,1) : take 2 (map f pi4)                                -- (-)
   = (1,1) : take 2 (map f ((1,1) : map f pi4))               -- M2
   = (1,1) : take 2 (f (1,1) : map f (map f pi4))             -- f
   = (1,1) : take 2 (f (1,1) : map f (map f pi4))             -- T2
   = (1,1) : f (1,1) : take (2-1) (map f (map f pi4))          -- f
   = (1,1) : (- 1,1+2) : take (2-1) (map f (map f pi4))       -- neg
   = (1,1) : (-1,1+2) : take (2-1) (map f (map f pi4))        -- (+)
   = (1,1) : (-1,3) : take (2-1) (map f (map f pi4))          -- (-)
   = (1,1) : (-1,3) : take 1 (map f (map f pi4))              -- pi4
   = (1,1) : (-1,3) : take 1 (map f (map f ((1,1) : map f pi4))) -- M2
   = (1,1) : (-1,3) : take 1 (map f (f (1,1) : map f (map f pi4))) -- M2
   = (1,1) : (-1,3) : take 1 (f f (1,1) : (map f (map f (map f pi4)))) -- T2
   = (1,1) : (-1,3) : f f (1,1) : take (1-1) (map f (map f (map f pi4))) -- F
   = (1,1) : (-1,3) : f (- 1,1+2) : take (1-1) (map f (map f (map f pi4))) -- F
   = (1,1) : (-1,3) : (- - 1,(1+2)+2) : take (1-1) (map f (map f (map f pi4))) -- neg
   = (1,1) : (-1,3) : (- -1,(1+2)+2) : take (1-1) (map f (map f (map f pi4))) -- neg
   = (1,1) : (-1,3) : (1,(1+2)+2) : take (1-1) (map f (map f (map f pi4))) -- (+)
   = (1,1) : (-1,3) : (1,3+2) : take (1-1) (map f (map f (map f pi4))) -- (+)
   = (1,1) : (-1,3) : (1,5) : take 0 (map f (map f (map f pi4)))           -- T1
   = (1,1) : (-1,3) : (1,5) : []                                         -- sucre sintàctic
   = [(1,1),(-1,3),(1,5)]
```

```

3. def pi4():
    num, den = 1, 1
    while True:
        yield num, den
        num, den = -num, den + 2
```

**Errors freqüents:**

- No utilitzar Haskell o Python
- No indicar els pasos

## 4 Conceptes avançats

1. 

```
def mymap(f, l, res=[]):
    if l == []:
        return res
    else:
        return mymap(f, l[1:], res + [f(l[0])])
```
2. El cost és  $O(n^2)$ , ja que afegir un element al final d'una llista amb l'operador `+` té cost lineal. No és una solució gens eficient!
3. BBA, pel *late binding*.
4. Si tenim  $x :: t$  (tipus),  $e :: s$  (tipus) i  $s \leq t$  (subtipus) aleshores  $x = e$  és correcte i  $e = x$  incorrecte.
5. 122
6. Una corutina és una funció asíncrona de la que podem aturar i reprendre la seva execució.

# Llenguatges de Programació, FIB, 14 de juny de 2024

*L'examen dura tres hores. Poseu el vostre nom a cada full. Contesteu cada problema enfulls diferents. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de lesrespostes. Feu bona lletra.*

## 1 $\lambda$ -càlcul (2 punts)

Recordeu que en  $\lambda$ -càlcul tenim aquestes equivalències:

<b>true</b>	$\equiv$	$\lambda x.\lambda y.x$
<b>false</b>	$\equiv$	$\lambda x.\lambda y.y$
<b>0</b>	$\equiv$	$\lambda s.\lambda z.z$
<b>1</b>	$\equiv$	$\lambda s.\lambda z.sz$
<b>2</b>	$\equiv$	$\lambda s.\lambda z.s(sz)$
<b>if</b>	$\equiv$	$\lambda c.\lambda x.\lambda y.cxy$
<b>leq</b>	$\equiv$	$\lambda m.\lambda n.\text{isZero}(\text{minus } m\ n)$
<b>isZero</b>	$\equiv$	$\lambda n.n(\lambda x.\text{false})\text{true}$
<b>minus</b>	$\equiv$	$\lambda m.\lambda n.n \text{ pred } m$

Per al cas de **pred**, utilitzarem la notació aritmètica per simplificar i escurçar la solució del problema. És a dir, assumirem que **pred 2**  $\equiv$  **1** i que **pred 0**  $\equiv$  **0**.

1. Digueu com farieu el **maxim** de dos nombres naturals utilitzant els combinadors anteriors.
2. Comproveu el resultat d'avaluar **maxim 2 1** en ordre normal tot explicitant cada operacions que apliqueu.

## 2 Avaluació mandrosa

(2 punts)

Considereu aquesta seqüència infinita definida recursivament:

$$rec = 0 : \text{scanl } (+) \ 1 \ rec \quad \text{-- (R)}$$

Aquestes són unes possibles definicions per a les funcions auxiliars que fa servir:

$$\begin{aligned} \text{scanl } _- \_ \ [ ] &= [ ] && \text{-- (S1)} \\ \text{scanl } f \ z \ (x:xs) &= z : \text{scanl } f \ (f \ z \ x) \ xs && \text{-- (S2)} \end{aligned}$$

$$\begin{aligned} \text{take } 0 \ _- &= [ ] && \text{-- (T1)} \\ \text{take } n \ (x:xs) &= x : \text{take } (n-1) \ xs && \text{-- (T2)} \end{aligned}$$

Contesteu les preguntes següents:

1. Mostreu (pas a pas) com Haskell evalua `take 3 rec` utilitzant evaluació mandrosa. Per fer-ho més curt, apliqueu les sumes ansiosament. A cada pas, indiqueu quina equació s'aplica. Podeu començar així:

$$\begin{aligned} \text{take } 3 \ rec &= \\ &= \text{take } 3 \ (0 : \text{scanl } (+) \ 1 \ rec) && \text{-- (R)} \\ &= \dots \end{aligned}$$

2. Què calcula `rec`?
3. Quin és el temps asimptòtic de l'avaluació de `take n rec`? Justifiqueu la vostra resposta.
4. Doneu una definició de la funció `rec` més eficient i digueu quin és el temps asimptòtic de `take n rec` amb la vostra definició.

### 3 Inferència de tipus

(2 punts)

Recordeu els tipus d'aquests operadors habituals:

$$\begin{aligned}(:) &:: a \rightarrow [a] \rightarrow [a] \\fmap &:: \text{Functor } f \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b \\<*> &:: \text{Applicative } f \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b\end{aligned}$$

A més, recordeu la definició de la classe **Applicative** i algunes de les seves instàncies:

```
class Functor f ⇒ Applicative f where
    pure :: a → f a
    <*> :: f (a → b) → f a → f b
```

```
instance Applicative Maybe where
    pure = Just
    Just f <*> Just x = Just (f x)
    _ <*> _ = Nothing
```

```
instance Applicative [ ] where
    pure x = [x]
    fs <*> xs = [f x | f ← fs, x ← xs]
```

Haskell conté una classe **Traversable** a la qual pertanyen totes les estructures de dades que poden ser recorregudes d'esquerra a dreta. Una de les seves funcions bàsiques és *sequenceA*:

```
instance Traversable Maybe where
    sequenceA Nothing = Nothing
    sequenceA (Just x) = fmap Just x
```

```
instance Traversable [ ] where
    sequenceA [] = pure []
    sequenceA (u:us) = (fmap (:)) u <*> (sequenceA us)
```

1. Calculeu (pas a pas) quan valen aquestes expressions:

- *sequenceA [Just 1, Just 2]*
- *sequenceA [Just 1, Nothing]*
- *sequenceA (Just [1, 2])*
- *sequenceA [[1, 2], [3, 4]]*

2. Inferiu el tipus complet de:

*sequenceA (u:us)*

## 4 Clausures en Python

(2 punts)

Una clausura és una funció que té accés a l'àmbit que la conté. A continuació teniu un exemple del seu ús en Python:

```
def exterior ( inici ):
    fi = '!'
    def interior (nom):
        print( inici , nom, fi )
    return interior

saluda = exterior ('Hola')
saluda('Pep')
```

Aquest codi imprimeix Hola Pep !.

També, en Python podem crear una funció que tingui un nombre indeterminat de paràmetres amb l'operador \* de la forma següent:

```
def aux(*args):
    print(type(args))
    print(args)
aux(1,2)
```

Aquest codi imprimeix <class 'tuple'> i (1, 2).

La llibreria *functools* conté una funció *partial* relacionada amb la currificació.

1. Doneu una implementació python de la funció *partial* que faci funcionar el codi següent fent que la sortida sigui 6:

```
multiplica = lambda x,y: x * y
doble = partial (multiplica , 2)
print(doble (3))
```

2. Doneu una implementació de la funció *partialN* que faci funcionar el codi següent fent que la sortida sigui respectivament 10 i 25:

```
from functools import reduce

def sumaRed(*args):
    return reduce(suma, list(args), 0)

print(sumaRed(1,2,3,4))
sumaRed10 = partialN(sumaRed, 1, 2, 3, 4)
print(sumaRed10(5, 5, 5))
```

## 5 Una IA m'explica Clojure i em posa un problema (2 punts)

 Bon dia!

 Bon dia, amb què et puc ajudar?

 Voldria tenir alguna noció de Clojure.

 Excel·lent! Clojure és un llenguatge de programació funcional basat en Lisp, dissenyat per ser simple i expressiu i que compila a la Java Virtual Machine.

 Dona'm alguns exemples senzills de Clojure tenint en compte que ja sé Haskell.

 I tant. Sabent Haskell no et costarà gens. Però tingues en compte que cal posar parèntesis al voltant de cada expressió. A continuació tens alguns exemples d'expressions en Closure amb la sortida després d'un comentari (punt i coma):

```
(zero? 1) ; false
(dec 67) ; 66
(mod 7 2) ; 1
(if (< 7 2) true false) ; false
;; funció anònima
((fn [x y] (+ x y)) 2 3) ; 5
;; let
(let [x 3] (+ x 2)) ; 5
```

 Com es definiria la funció factorial en Clojure?

 La funció factorial s'escriu naturalment de forma recursiva en Clojure:

```
(defn factorial [n]
  (if (zero? n)
    1
    (* n (factorial (dec n)))))
```

 Però... amb un nombre molt gran, no hi ha perill de sobreeiximent de la pila?

 En efecte, en treballar amb nombres grans, aquesta funció té problemes amb l'espai de la pila. Una possible solució consistiria a utilitzar l'optimització de la recursivitat per la cua (*tail call optimization*), però la JVM no la suporta directament. Tanmateix, Clojure ofereix una funció **recur** per poder-la aplicar: Quan s'utilitza la funció **recur** al final de la funció recursiva, el compilador activa l'optimització, que fa que no s'incrementi la mida de la pila:

```
(defn factorial2 [n acc]
  (if (zero? n)
    acc
    (recur (dec n) (* n acc))))
```

```
(defn factorial [n] (factorial2 n 1))
```

 Hauria de mirar-ho bé! Quin exercici em proposaries per practicar?

 Un exercici senzill seria escriure una funció **prime?** en Clojure que indiqui si un nombre natural és un nombre primer o no. Assegura't d'utilitzar la funció **recur** per evitar sobreeiximents de la pila. Fixa't que no et cal saber com es calculen arrels quadrades en Clojure.

# Llenguatges de Programació, FIB, 12 de gener de 2024

## Possible solució

### 1 $\lambda$ -càlcul

1. **maxim**  $\equiv \lambda a. \lambda b. \text{if} (\text{leq } a\ b) \ b\ a$

2.

```
maxim 2 1
(\lambda a. \lambda b. \text{if} (\text{leq } a\ b) \ b\ a) 2 1
\text{if} (\text{leq } 2\ 1) 1\ 2
(\lambda c. \lambda x. \lambda y. cxy) (\text{leq } 2\ 1) 1\ 2
\text{leq } 2\ 1\ 1\ 2
(\lambda m. \lambda n. \text{isZero} (\text{minus } m\ n)) 2\ 1) 1\ 2
\text{isZero} (\text{minus } 2\ 1) 1\ 2
(\lambda n. n (\lambda x. \text{false}) \text{true}) (\text{minus } 2\ 1) 1\ 2
\text{minus } 2\ 1 (\lambda x. \text{false}) \text{true } 1\ 2
(\lambda m. \lambda n. n \text{ pred } m) 2\ 1 (\lambda x. \text{false}) \text{true } 1\ 2
1 \text{ pred } 2 (\lambda x. \text{false}) \text{true } 1\ 2
(\lambda s. \lambda z. sz) \text{ pred } 2 (\lambda x. \text{false}) \text{true } 1\ 2
\text{pred } 2 (\lambda x. \text{false}) \text{true } 1\ 2
1 (\lambda x. \text{false}) \text{true } 1\ 2
(\lambda s. \lambda z. sz) (\lambda x. \text{false}) \text{true } 1\ 2
(\lambda z. (\lambda x. \text{false}) z) \text{true } 1\ 2
(\lambda x. \text{false}) \text{true } 1\ 2
\text{false } 1\ 2
(\lambda x. \lambda y. y) 1\ 2
(\lambda y. y) 2
```

definició de **maxim**  
 $\beta$ -reducció de *a* i *b*  
definició de **if**  
 $\beta$ -reducció de *c*, *x* i *y*  
definició de **leq**  
 $\beta$ -reducció de *m* i *n*  
definició de **isZero**  
 $\beta$ -reducció de *n*  
definició de **minus**  
 $\beta$ -reducció de *m* i *n*  
definició de 1  
 $\beta$ -reducció de *s* i *z*  
aplicació de **pred**  
definició de 1  
 $\beta$ -reducció de *s*  
 $\beta$ -reducció de *z*  
 $\beta$ -reducció de *x*  
definició de **false**  
 $\beta$ -reducció de *x*  
 $\beta$ -reducció de *y*  
q.e.d.

### Errors freqüents:

- Precedència de les operacions
- Poca rigurositat en l'aplicació de les  $\beta$ -reduccions

## 2 Avaluació mandrosa

```

1. take 3 rec =
   = take 3 (0 : scanl (+) 1 rec)                                -- (R)
   = 0 : take 2 (scanl (+) 1 rec)                                -- (T2)
   = 0 : take 2 (scanl (+) 1 (0 : scanl (+) 1 rec))           -- (R)
   = 0 : take 2 (1 : scanl (+) 1 (scanl (+) 1 rec))          -- (S2)
   = 0 : 1 : take 1 (scanl (+) 1 (scanl (+) 1 rec))          -- (R)
   = 0 : 1 : take 1 (scanl (+) 1 (scanl (+) 1 (0 : scanl (+) 1 rec)))  -- (S2)
   = 0 : 1 : take 1 (scanl (+) 1 (1 : scanl (+) 1 (scanl (+) 1 rec)))  -- (S2)
   = 0 : 1 : take 1 (1 : scanl (+) 2 (scanl (+) 1 (scanl (+) 1 rec)))  -- (S2)
   = 0 : 1 : 1 : take 0 (scanl (+) 2 (scanl (+) 1 (scanl (+) 1 rec)))  -- (T2)
   = 0 : 1 : 1 : []                                                 -- (T1)
   = [0, 1, 1]                                                    -- sucre sintàctic

```

2. *rec* és la llista infinita dels nombres de Fibonacci.
3. El temps asimptòtic de l'avaluació de **take** *n* *rec* és  $O(n^2)$  ja que calen *n* aplicacions de **take** i cadascuna d'elles necessita un **scanl** més que l'anterior.
4. Amb el codi següent el cost de **take** *n* *rec* és  $O(n)$  ja que cada element de la llista es calcula en temps constant respecte de l'anterior:

```

rec = map fst $ iterate seg (0,1)
      where seg (a, b) = (b, a+b)

```

### 3 Inferència de tipus

1.

•

```
sequenceA [Just 1, Just 2] =  
fmap (:) (Just 1) <*> sequenceA [Just 2] =  
Just (1:) <*> sequenceA [Just 2] =  
Just (1:) <*> (fmap (:) (Just 2) <*> sequenceA []) =  
Just (1:) <*> (Just (2:) <*> sequenceA []) =  
Just (1:) <*> (Just (2:) <*> Just []) =  
Just (1:) <*> Just (2:[]) =  
Just (1:) <*> Just [2] =  
Just (1:[2])  
Just [1, 2]
```

•

```
sequenceA [Just 1, Nothing] =  
fmap (:) (Just 1) <*> sequenceA [Nothing] =  
Just (1:) <*> sequenceA [Nothing] =  
Just (1:) <*> (fmap (:) Nothing <*> sequenceA []) =  
Just (1:) <*> (Nothing <*> sequenceA []) =  
Just (1:) <*> Nothing =  
Nothing
```

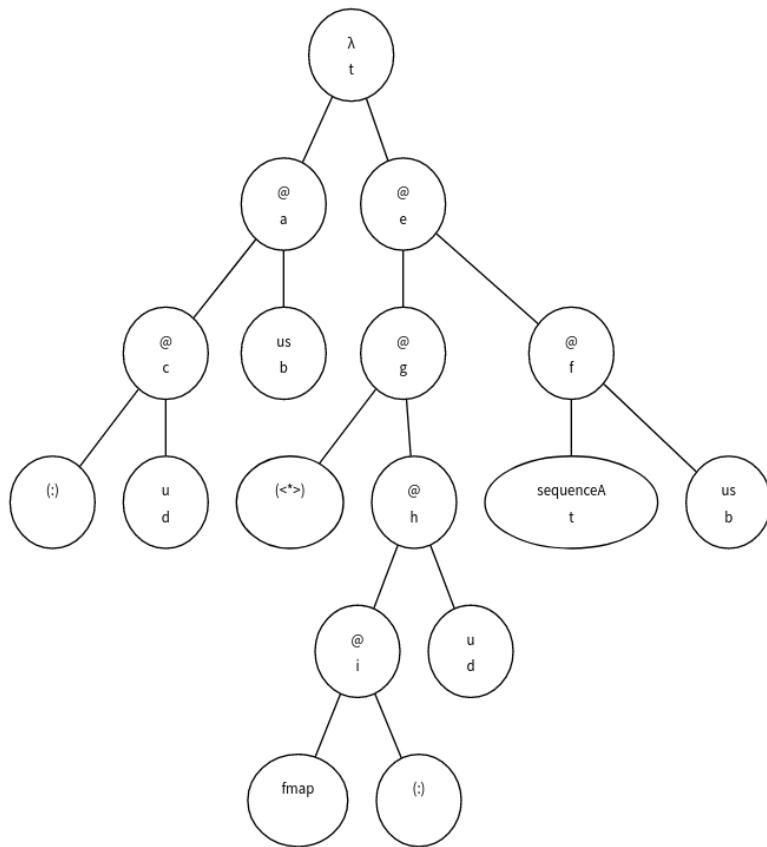
•

```
sequenceA $ Just [1, 2] =  
fmap Just [1, 2] =  
[Just 1, Just 2]
```

•

```
sequenceA [[1, 2], [3, 4]] =  
fmap (:) [1, 2] <*> sequenceA [[3, 4]] =  
[(1:), (2:)] <*> sequenceA [[3, 4]] =  
[(1:), (2:)] <*> (fmap (:) [3, 4] <*> sequenceA []) =  
[(1:), (2:)] <*> ([[3:], (4:)]) <*> sequenceA [] =  
[(1:), (2:)] <*> ([[3:], (4:)]) <*> [[]] =  
[(1:), (2:)] <*> [[3], [4]] =  
[[1,3],[1,4],[2,3],[2,4]]
```

2.



- (i)  $\text{fmap} :: \text{Functor } f1 \Rightarrow (a1 \rightarrow b1) \rightarrow f1 \ a1 \rightarrow f1 \ b1$   
 $(:) :: a2 \rightarrow [a2] \rightarrow [a2]$   
 $a1 = a2$   
 $b1 = [a2] \rightarrow [a2]$   
 $i = f1 \ a2 \rightarrow f1 \ ([a2] \rightarrow [a2])$
- (h)  $d = f1 \ a2$   
 $h = f1 \ ([a2] \rightarrow [a2])$
- (g)  $<*> :: \text{Applicative } f3 \Rightarrow f3 \ (a3 \rightarrow b3) \rightarrow f3 \ a3 \rightarrow f3 \ b3$   
 $f1 = f3$   
 $a3 = b3 = [a2]$   
 $g = f1 \ [a2] \rightarrow f1 \ [a2]$
- (c)  $(:) :: a4 \rightarrow [a4] \rightarrow [a4]$   
 $d = a4 = f1 \ a2$   
 $c = [f1 \ a2] \rightarrow [f1 \ a2]$
- (a)  $b = a = [f1 \ a2]$
- (t)  $g = f1 \ [a2] \rightarrow f1 \ [a2]$   
 $t = [f1 \ a2] \rightarrow e$   
 $f \rightarrow e = f1 \ [a2] \rightarrow f1 \ [a2]$   
 $e = f = f1 \ [a2]$   
 $t = [f1 \ a2] \rightarrow f1 \ [a2]$

**Solució:**  $\text{Applicative } f \Rightarrow [f \ a] \rightarrow f[a]$

## 4 Clausures en Python

```
def partial (f, x):
    def g(*args):
        return f(*((x,) + args))
    return g

def partialN (*args):
    def g(*args2):
        f = args[0]
        xs = args [1:] + args2
        return f(*xs)
    return g
```

## 5 Una IA m'explica Clojure i em posa un problema

```
(defn prime? [n]
  (let [find-divisor
        (fn [i n]
          (if (> i (/ n 2)) false
              (if (zero? (mod n i)) true
                  (recur (inc i) n))))]
    (if (< n 2) false
        (not (find-divisor 2 n)))))
```

# Llenguatges de Programació, FIB, 12 de gener de 2024

*L'examen dura tres hores. Poseu el vostre nom a cada full. Contesteu cada problema en fulls diferents. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de les respostes. Feu bona lletra.*

## 1 $\lambda$ -càcul (1'5 punts)

1. Doneu una definició de la funció **flip** de Haskell en  $\lambda$ -càcul.
2. Recordeu que en  $\lambda$ -càcul:

$$\begin{aligned}\mathbf{true} &\equiv \lambda xy.x \\ \mathbf{false} &\equiv \lambda xy.y \\ \mathbf{two} &\equiv \lambda sz.s(z)\end{aligned}$$

Donada la definició

$$\mathbf{even} \equiv \lambda n. n \mathbf{flip} \mathbf{true}$$

comproveu que el resultat d'avaluar **even two** en ordre normal és **true**.

## 2 Raonament equacional en Haskell (2 punts)

Recordeu que la funció **foldl** ve donada per:

$$\begin{aligned}\mathbf{foldl}\ f\ e\ [] &= e \\ \mathbf{foldl}\ f\ e\ (x:xs) &= \mathbf{foldl}\ f\ (f\ e\ x)\ xs\end{aligned}$$

1. Doneu una funció *fact1* que calculi el factorial d'un natural *n* de forma recursiva.
2. Doneu una funció *fact2* que calculi el factorial d'un natural *n* amb el **foldl**.
3. Doneu (sense necessitat de demostrar) una equació que relacioni les expressions següents:
  - **foldl** *f e [1..n]*
  - **foldl** *f e [1..n-1]*.
4. Demostreu per raonament equacional que *fact1* i *fact2* són equivalents. Podeu utilitzar la relació del punt anterior com a lema.

### 3 Inferència de tipus

(2 punts)

Recordeu els tipus d'aquestes funcions habituals:

```
id  :: a → a  
const  :: a → b → a  
(.)  :: (b → c) → (a → b) → a → c
```

A més, recordeu que les classes *Functor* i *Applicative* venen definides per

```
class Functor f where  
    fmap :: (a → b) → (f a → f b)  
  
class Functor f ⇒ Applicative f where  
    pure :: a → f a  
    <*>  :: f (a → b) → f a → f b
```

També, recordeu que *Maybe* instancia aquestes classes així:

```
instance Functor Maybe where  
    fmap f Nothing = Nothing  
    fmap f (Just x) = Just (f x)  
  
instance Applicative Maybe where  
    pure = Just  
    Nothing <*> _ = Nothing  
    Just f <*> x = fmap f x
```

La llibreria estàndard de Haskell defineix l'operador `<$` i l'operador `*>` d'aquesta forma:

```
(<$)    = fmap . const  
x *> y = (id <$ x) <*> y
```

1. Calculeu (pas a pas) quan valen aquestes expressions:

- `'a' <$ (Just 2)`
- `'a' <$ Nothing`
- `(Just 'a') *> (Just 2)`

2. Inferiu el tipus complet de l'operador `<$`. Per a fer-ho, dibuixeu l'arbre de sintaxi abstracta corresponent, anoteu l'arbre amb els tipus de cada node, genereu el conjunt de restriccions d'igualtat de tipus i resoleu les restriccions. No us deixeu els requeriments sobre les classes.

3. Igualment, inferiu el tipus complet de l'operador `*>`.

## 4 Python amb tipus

(2 punts)

Malgrat que Python és un llenguatge de tipus dinàmic, des de fa un temps ofereix la possibilitat d'anotar els tipus de les variables, paràmetres i funcions. En temps d'execució, aquestes anotacions no provoquen cap impacte en la manera que s'executen els programes ni sobre el seu temps d'execució: simplement es consideren com a comentaris. L'objectiu d'aquestes anotacions de tipus és permetre que altres eines, com ara `mypy`, puguin verificar estàticament l'absència d'errors de tipus.

Per exemple, en el següent programa en Python amb tipus, `mypy` diagnostica que la crida a factorial és incorrecta abans d'executar el programa (estalviant així un error de tipus en temps d'execució):

```
def factorial (n: int) → int: ...
print( factorial ([1,2,3]))
```

El tipus de les funcions amb paràmetres de tipus  $T_1, T_2, \dots$  i resultat de tipus  $R$  s'escriu `Callable[[ $T_1, T_2, \dots$ ],  $R$ ]`. Per exemple, aquesta seria la capçalera d'una funció `foldl`:

```
def foldl (Callable[[ $B, A$ ],  $B$ ],  $B$ , list [ $A$ ]) →  $B$ : ...
```

on  $A$  i  $B$  són variables de tipus (iguals que en Haskell).

Considereu aquest fragment de programa:

```
class Persona: ...
class Empleat(Persona): ...
class Mecanic(Empleat): ...

def f_persona(p: Persona) → int: ...
def f_empleat(e: Empleat) → int: ...
def f_mecanic(m: Mecanic) → int: ...

def g_persona(i: int) → Persona: ...
def g_empleat(i: int) → Empleat: ...
def g_mecanic(i: int) → Mecanic: ...

def h1(f: Callable[[Empleat], int]) → None: ...
def h2(g: Callable[[int], Empleat]) → None: ...
```

Llavors, si s'escriu

```
h1(f_persona)      # 1
h1(f_empleat)     # 2
h1(f_mecanic)     # 3 !

h2(g_persona)      # 4 !
h2(g_empleat)     # 5
h2(g_mecanic)     # 6
```

`mypy` dóna un error de tipus a la línia 3 i un error de tipus a la línia 4.

Utilitzant la informació anterior, contesteu les preguntes següents:

1. Doneu les definicions de constructor de tipus covariant, contravariant i invariant.
2. Digueu si, per a `mypy`, el constructor de tipus **Callable** és covariant, contravariant i/o invariant *en els paràmetres* i expliqueu perquè.
3. Digueu si, per a `mypy`, el constructor de tipus **Callable** és covariant, contravariant i/o invariant *en el resultat* i expliqueu perquè.
4. Expliqueu per quina raó té sentit l'error de la línia 3.
5. Expliqueu per quina raó té sentit l'error de la línia 4.

## 5 Aprenent Scheme amb una IA

(2'5 punts)

Considereu aquesta conversa real (minimament editada) entre una usuària i ChatGPT:

 Hola màquina!

 Bon dia! Com puc ajudar-te avui?

 Voldria aprendre a programar en Scheme.

 Genial! Scheme és un llenguatge de programació funcional fascinant. Scheme és un llenguatge de programació funcional i minimalist, derivat de Lisp. Es destaca per la seva elegància i simplicitat, amb una sintaxi simple basada en parèntesis que facilita la manipulació d'expressions i el tractament de funcions com a valors.

 Està bé. Ensenya'm una funció en Scheme que calculi el màxim de dos nombres.

 Clar, aquí tens un exemple de funció que calcula el màxim de dos números en Scheme:

```
(define (maxim a b)      ; funció que retorna el més gran de dos valors
  (if (> a b)
      a
      b))
```

 I com escriuries una funció en Scheme per saber si un nombre és primer o no?

 Per determinar si un nombre és primer o no en Scheme, pots escriure una funció que verifiqui si té més d'un divisor a part d'1 i d'ell mateix. Aquí tens una manera de fer-ho:

```
(define (es-primer? n)           ; n és un natural
  (define (divisible? divisor)
    (= (remainder n divisor) 0))

  (define (troba-divisor divisor)
    (if (> divisor (/ n 2))        ; si el divisor arriba a n/2,
        #t                         ; n és primer.
        (if (divisible? divisor)   ; sinó, si té un divisor,
            #f                         ; n no és primer.
            (troba-divisor (+ divisor 1))))); sinó, continua buscant

  (troba-divisor 2))
```

 Explica'm com es crea una llista i com s'afegeix un element al principi d'una llista.

 En Scheme, la llista buida s'escriu '(). La funció `cons` s'utilitza per construir llistes afegint un element al davant d'una altra llista. Per exemple, per crear una llista amb els elements 1, 2 i 3, pots fer-ho així:

```
(define una-llista (cons 1 (cons 2 (cons 3 '()))))
```

 Existeixen operacions com `head` i `tail` de Haskell?

 Sí: `head` es diu `car` i `tail` es diu `cdr`.

 Moltes gràcies.

 De res, estic aquí per ajudar! Espero que l'examen d'LP et vagi molt bé.

Contesteu les preguntes següents:

1. La sintaxi de Scheme es basa, essencialment, en expressions parentitzades. Des del punt de vista de la pragmàtica dels llenguatges de programació, doneu un argument a favor i un argument en contra d'aquesta decisió de disseny.
2. Escriviu, en format ANTLR, una gramàtica per reconèixer programes en Scheme amb els elements donats als exemples.
3. La funció *divisible?* dins de la funció *es-primer?* utilitza la variable *n* de *es-primer*. Com s'anomena aquest concepte als llenguatges de programació?
4. Quin és l'espai (asimptòtic) utilitzat per la funció *es-primer??* Expliqueu perquè.
5. És correcta la implementació de *es-primer?* donada per ChatGPT?
6. Escriviu en Scheme una funció *range* que, donats dos enters *i* i *j*, retorna la llista dels enters de *i* a *j*.
7. Escriviu en Scheme una funció *filter* que, donat un predicat i una llista, retorna una llista amb els elements de la llista donada que compleixen el predicat.
8. Escriviu en Scheme una funció *primers-primers* que, donat un natural *n*, retorna la llista de tots els nombres primers fins a *n* (inclòs). No cal que sigui eficient però sí elegant.

# Llenguatges de Programació, FIB, 12 de gener de 2024

## Possible solució

### 1 $\lambda$ -càlcul

1. flip $\equiv \lambda xyz.xzy$	
2. even $\equiv \lambda n.n$	definició de even
flip true	$\beta$ -reducció de $n$
even two	definició de two
( $\lambda n.n$ flip true) two	$\beta$ -reducció de $s$
two flip true	$\beta$ -reducció de $z$
( $\lambda sz.s(sz)$ ) flip true	definició de flip
( $\lambda z.(\text{flip } (\text{flip } z))$ ) true	$\beta$ -reducció de $x$
flip(flip true)	definició de flip
$\lambda xyz.xzy$ (flip true)	$\beta$ -reducció de $x$
$\lambda yz.((\text{flip true})zy)$	definició de flip
$\lambda yz.(((\lambda xyz.xzy) \text{ true})zy)$	$\beta$ -reducció de $x$
$\lambda yz.(\lambda yz.(\text{true } zy)zy)$	$\alpha$ -conversió [ $y/t$ ]
$\lambda yz.(\lambda tz.(\text{true } zt)zy)$	$\alpha$ -conversió [ $z/u$ ]
$\lambda yz.(\lambda tu.(\text{true } ut)zy)$	$\beta$ -reducció de $t$
$\lambda yz.(\lambda u.(\text{true } uz)y)$	$\beta$ -reducció de $u$
$\lambda yz.(\text{true } yz)$	definició de true
$\lambda yz.((\lambda xy.x)yz)$	$\alpha$ -conversió [ $x/t$ ]
$\lambda yz.((\lambda ty.t)yz)$	$\alpha$ -conversió [ $y/u$ ]
$\lambda yz.((\lambda tu.t)yz)$	$\beta$ -reducció de $t$
$\lambda yz.((\lambda u.y)z)$	$\beta$ -reducció de $u$
$\lambda yz.y$	$\alpha$ -equivalència
true	

### Errors freqüents:

- Considerar que el flip només té dos paràmetres i fer  $\lambda xy.yx$
- Precedència de les operacions
- Poca rigurositat en l'aplicació de les  $\beta$ -reduccions

## 2 Raonament equacional en Haskell

(2 punts)

1. Recursiu:

$$\begin{array}{ll} \text{fact1 } 1 = 1 & \text{-- R1} \\ \text{fact1 } n = n * \text{fact1 } (n-1) & \text{-- R2} \end{array}$$

2. Fold:

$$\text{fact2} = \text{foldl } (*) \ 1 \ [1..n] \quad \text{-- R3}$$

3. Lema:

$$\text{foldl } f \ e \ [1..n] = f (\text{foldl } f \ e \ [1..n-1]) n \quad \text{-- R4}$$

4. Demostració de  $\text{fact1 } n = \text{fact2 } n$  per inducció:

Cas base: $\boxed{\begin{array}{ c c } \hline \text{fact1 } 1 & \text{R1} \\ \hline 1 & \\ \hline \end{array}}$	$\begin{array}{l} \text{fact2 } 1 \\ \text{foldl } (*) \ 1 \ [1..1] \\ \text{foldl } (*) \ 1 \ [1] \\ (*) \ 1 \ 1 \\ 1 \end{array}$	R3 rang funcionament de foldl multiplicació
---	---	--

$$\text{H.I.: fact1 } (n-1) = \text{fact2 } (n-1)$$

Cas inductiu: $\boxed{\begin{array}{ c c } \hline \text{fact1 } n & \text{R2} \\ \hline n * \text{fact1 } (n-1) & \\ \hline \end{array}}$	$\begin{array}{l} \text{fact2 } n \\ \text{foldl } (*) \ 1 \ [1..n] \\ (*) (\text{foldl } (*) \ 1 \ [1..n-1]) n \\ (*) (\text{fact1 } (n-1)) n \\ (*) n (\text{fact1 } (n-1)) \\ n * \text{fact1 } (n-1) \end{array}$	R3 R4 H.I. comutativitat de * a infixe
---	---	--

q.e.d.

Errors freqüents:

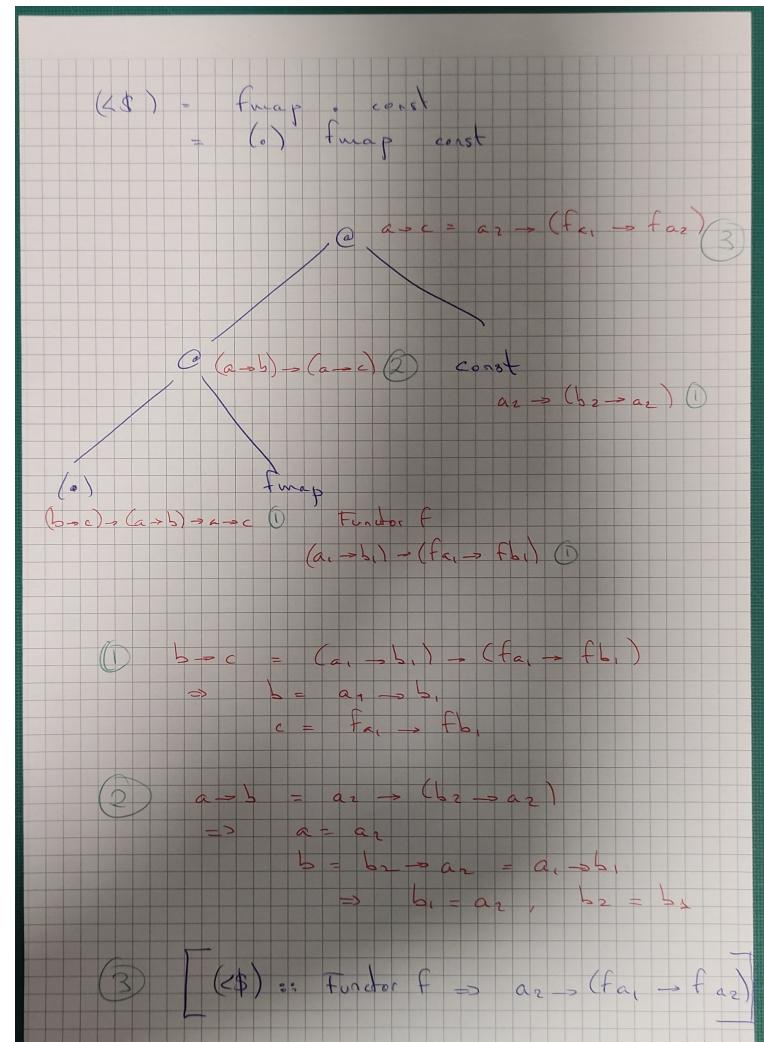
- Poca rigurositat en el raonament equacional
- Absència d'indicació de quina regla s'aplica en el raonament equacional

### 3 Inferència de tipus

$$\begin{aligned}
 1. \quad & 'a' <\text{Nothing} = (<) 'a' (\text{Just } 2) \\
 & = (\text{fmap . const}) 'a' (\text{Just } 2) \\
 & = \text{fmap} (\text{const } 'a') (\text{Just } 2) \\
 & = \text{Just} (\text{const } 'a' 2) \\
 & = \text{Just } 'a'
 \end{aligned}$$

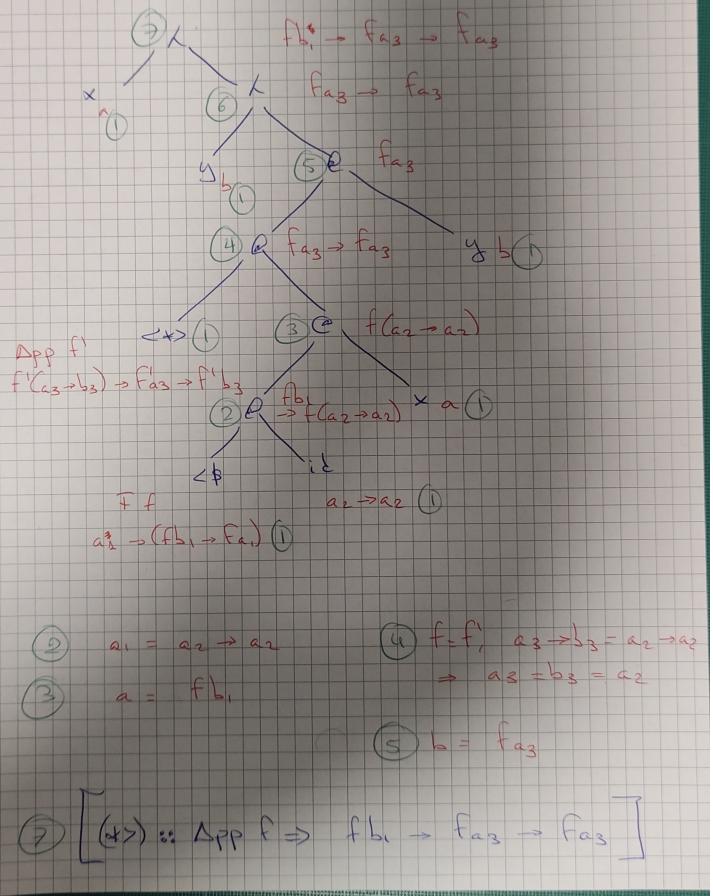
$$\begin{aligned}
 & 'a' <\text{Nothing} = (<) 'a' \text{ Nothing} \\
 & = (\text{fmap . const}) 'a' \text{ Nothing} \\
 & = \text{fmap} (\text{const } 'a') \text{ Nothing} \\
 & = \text{Nothing}
 \end{aligned}$$

$$\begin{aligned}
 & (\text{Just } a) * > (\text{Just } 2) \\
 & = (\text{id} <* \text{Just } 'a') <*> (\text{Just } 2) \\
 & = \text{fmap} (\text{const id}) (\text{Just } 'a') <*> (\text{Just } 2) \\
 & = \text{Just} (\text{const id}) 'a' <*> (\text{Just } 2) \\
 & = \text{Just id} <*> (\text{Just } 2) \\
 & = \text{fmap id} (\text{Just } 2) \\
 & = \text{Just} (\text{id } 2) \\
 & = \text{Just } 2
 \end{aligned}$$



2.

$(\leftrightarrow) = \lambda x y \rightarrow (i_b \leftrightarrow x) \leftrightarrow y$



3.

## 4 Python amb tipus

(2 punts)

1. Sigui  $s$  un subtipus de  $t$ :  $s \leq t$ . Sigui  $C$  un constructor de tipus.

- $C$  és covariant quan  $C[s] \leq C[t]$ .
- $C$  és contravariant quan  $C[t] \leq C[s]$ .
- $C$  és invariant quan no es té  $C[s] \leq C[t]$  ni  $C[t] \leq C[s]$ .

2. Tenim  $Mecanic \leq Empleat \leq Persona$ .

**Callable**`[[Empleat], int]` no dóna problemes amb *Empleat* ni *Persona*, però sí amb *Mecanic*.

Així, tenim **Callable**`[[Persona], int] \leq Callable[[Empleat], int]` però, en canvi, tenim **Callable**`[[Persona], int] \not\leq Callable[[Mecanic], int]`.

Per tant, **Callable** no és covariant però sí és contravariant en el tipus del paràmetre.

3. Tenim  $Mecanic \leq Empleat \leq Persona$ .

**Callable**`[[int], Empleat]` no dóna problemes amb *Empleat* ni *Mecanic*, però sí amb *Persona*.

Així, tenim **Callable**`[[int], Persona] \leq Callable[[int], Empleat]` però, en canvi, tenim **Callable**`[[int], Persona] \not\leq Callable[[int], Mecanic]`.

Per tant, **Callable** no és contravariant però és covariant en el tipus del resultat.

4.  $h1$  espera que la funció  $f$  accepti un *Empleat*. Si  $f$  acceptés un subtipus d'*Empleat* (com ara *Mecanic*),  $f$  podria no funcionar correctament. Per exemple:

```
def f_mecanic(m: Mecanic) → int:
    return m.cambiarOli()

def h1(f: Callable[[Empleat], int]) → None:
    e = Empleat()
    f(e)
```

$h1(f\_mecanic)$  # error: aplicarà *cambiarOli* (específic de *Mecanic*) a un *Empleat*

Per evitar aquest tipus d'error, té sentit que **Callable** no sigui covariant en el tipus del paràmetre.

5.  $h2$  espera que la funció  $g$  retorni un *Empleat*. Si  $g$  retornés un supertipus d'*Empleat* (com ara *Persona*),  $h2$  no el podria continuar processant correctament. Per exemple:

```
def g_persona(x: int) → Persona:
    return Persona()

def h2(g: Callable[[int], Empleat]) → None:
    e = g(42)
    e.cobrar_sou()
```

$h2(g\_persona)$  # error: aplicarà *cobrarSou* (específic d'*Empleat*) a una *Persona*

Per evitar aquest tipus d'error, té sentit que **Callable** no sigui contravariant en el tipus del resultat.

**Errors freqüents:**

- No saber identificar la covariància i la contravariància.
- Argument cíclic entre els apartats 2 i 4 i entre els apartats 3 i 5: "Com que dóna error és co/contra-variant, i com que és co/contra-variant dóna error."

## 5 Aprenent Scheme amb una IA

1. Un avantatge de la notació amb parèntesis és la seva simplicitat i uniformitat. Un desavantatge és que pot resultar difícil de llegir.

2. grammar scheme;

```
root: expressions;

expressions: expressio*;

expressio: atom | '(' expressions ')';

atom: OP | NUM | ID | CONST;

OP: '+' | '-' | '*' | '/'; // etc
NUM: [0-9]+;
ID: [a-zA-Z?]-]+;
CONST: '#t' | '#f' | '\''();';
WS: [ \t\r\n]+ -> skip;
COMMENT: ';' .*? '\n' -> skip;
```

Fixeu-vos que no cal fer definicions *ad hoc* per a defines, ifs, etc: Tot és uniforme.

**Errors freqüents:** Fer una gramàtica molt complicada quan no cal. Tenir en compte l'espaiat.

3. És una clausura.

4. Com que ha d'anar apilant les  $n/2$  crides, l'espai és  $O(n)$ . [Ara bé, com que és una funció recursiva final, el compilador pot optimitzar-la i convertir-la en una funció iterativa i l'espai és  $O(1)$ .]

5. La implementació de *es-primer?* donada per ChatGPT no és correcta del tot perquè no funciona amb els valors 0 i 1.

```
6. (define (range i j)
  (if (> i j)
      '()
      (cons i (range (+ i 1) j))))
```

```
7. (define (filter p l)
  (if (= l '())
      '()
      (if (p (car l))
          (cons (car l) (filter p (cdr l)))
          (filter p (cdr l)))))
```

```
8. (define (primers-primers n)
  (filter es-primer? (range 2 n)))
```

**Errors freqüents:** Usar *car* o *cdr* com a operacions mutadores enlloc de funcions consultores. Fer complicades les coses senzilles.

# Llenguatges de Programació, FIB, 21 de juny de 2023

*L'examen dura tres hores. Poseu el vostre nom a cada full. Contesteu cada problema enfulls diferents. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de lesrespostes. Feu bona lletra.*

## 1 $\lambda$ -càcul en Python amb una IA (2 punts)

Considereu aquest diàleg (minimament editat) entre una usuària i ChatGPT:

-  Bon dia!
-  Bon dia, amb què et puc ajudar?
-  Ajuda'm amb la pràctica d'LP.
-  Estaré encantat d'ajudar-te amb la pràctica d'LP (Llenguatges de Programació).
-  Explica'm què és una beta reducció en el lambda càcul.
-  Una beta reducció en el lambda càcul és una operació que permet avaluarexpressions lambda mitjançant la substitució de les ocurrences d'una variable lligada en una funció per l'argument corresponent.
-  Explica'm on cal aplicar una beta reducció a una lambda expressió en l'ordre normal.
-  En l'ordre normal, cal aplicar una beta reducció a una lambda expressió en l'expressió més externa que encara no ha estat reduïda.
-  Escriu-me en Python una funció

```
def normal_eval(e: Expr) -> Expr
```

que apliqui una  $\beta$ -reducció en ordre normal a una expressió en lambda càcul utilitzant els tipus següents:

```
@dataclass
class Leaf:
    var: str

@dataclass
class App:
    lft: Expr
    rgt: Expr

@dataclass
class Abs:
    var: str
    body: Expr

Expr = Leaf | App | Abs
```

Pots suposar que no cal aplicar cap  $\alpha$ -conversió.

 Aquí tens una possible implementació de la funció *normal\_eval* en Python que aplica una  $\beta$ -reducció en ordre normal d'una expressió en el càlcul lambda suposant que no cal aplicar cap  $\alpha$ -conversió:

```
def normal_eval(e: Expr) -> Expr:
    match e:
        case Leaf(var):
            return e
        case App(lft, rgt):
            if isinstance(lft, Abs):
                return substitute(lft.body, lft.var, rgt)
            else:
                return App(normal_eval(lft), normal_eval(rgt))
        case Abs(var, body):
            return Abs(var, normal_eval(body))

def substitute(e: Expr, var: str, r: Expr) -> Expr:
    match e:
        case Leaf(v):
            if v != var:
                return e
            else:
                return r
        case App(lft, rgt):
            return App(substitute(lft, var, r), substitute(rgt, var, r))
        case Abs(v, body):
            return Abs(v, substitute(body, var, r))
```

 Gràcies.

 De res! Estic aquí per ajudar-te.

Doneu la vostra opinió humana i experta. Màxim suggerit: 12 línies.

## 2 APL

(2 punts)

APL és un llenguatge de programació d'arrays altament expressiu i concís, dissenyat als anys 1960s per a operacions matemàtiques. Una de les pecularitats d'APL és l'ús de caràcters i símbols no habituals. Per exemple, el producte s'escriu  $\times$  i no  $*$  i la divisió  $\div$  enlloc de  $/$ . També, l'ordre d'avaluació en APL és de dreta a esquerra, si no s'usen parèntesis. Per tant,  $5 \times 6 + 7$  val 65, però  $(5 \times 6) + 7$  val 37.

Aquest programa en APL calcula tots els nombres primers fins a 100:

```
R ← 100  
( R ∈ R o . × R ) / R ← 1 ↓ ↴ R
```

Entre d'altres, el programa anterior utilitza la funció  $\iota$  (iota, una i sense punt) que retorna un vector amb els naturals d'1 fins al seu paràmetre (així,  $\iota R$  retorna  $[1, 2, 3, \dots, R]$ ).

Aquesta expressió en APL

```
⌈ / v
```

calcula el màxim d'un vector  $v$ . De forma semblant, aquesta expressió en APL

```
+ / v
```

calcula la suma dels elements d'un vector  $v$ .

A més, la funció  $\#$  davant d'un vector en retorna la seva mida.

L'operador  $\approx$  (que recorda la mirada confosa d'un neòfit en APL) funciona així:  $5 - 3$  val 2 i  $8 \div 2$  val 4 però  $5 - \approx 3$  val -2 i  $8 \div \approx 2$  val 0.25.

Tenint en compte les informacions anteriors:

1. Quines funcions representen els operadors binaris  $\lceil$ ,  $/$  i  $\approx$  en APL?
2. Quins serien els seus possibles equivalents en Haskell i en Python?
3. Escriviu una expressió en APL que calculi

$$2 \max(a, \frac{3}{b+c}) - 1$$

sense utilitzar parèntesis.

4. Escriviu una expressió en APL que calculi el factorial d'un nombre  $n$ .
5. Escriviu una expressió en APL que calculi la mitjana d'un vector  $v$ .

### 3 Inferència de tipus

(2 punts)

Inferiu el tipus complet de la funció

*mySort = sortBy (compare `on` fst )*

sabent que

```
data Ordering = LT | EQ | GT
sortBy :: (a → a → Ordering) → [a] → [a]
compare :: Ord a ⇒ a → a → Ordering
on :: (b → b → c) → (a → b) → a → a → c
fst :: (a,b) → a
```

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

## 4 Raonament equacional

(2 punts)

Aquestes són algunes funcions predefinides de Haskell:

**replicate** :: **Int** →  $a \rightarrow [a]$

**replicate** 0  $x = []$

**replicate**  $n x = x : \text{replicate} (n - 1) x$

(R1)

(R2)

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

**foldr**  $f z [] = z$

**foldr**  $f z (x:xs) = f x (\text{foldr} f z xs)$

(F1)

(F2)

I aquesta és la definició de la classe mònada:

**class Monad**  $m$  **where**

$(\gg=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$

$(\gg) :: m a \rightarrow m b \rightarrow m b$

**return** ::  $a \rightarrow m a$

$x \gg y = x \gg= \lambda \_ \rightarrow y$

(D)

que té aquestes lleis:

**return**  $x \gg= f = f x$

(L1)

$m \gg= \text{return} = m$

(L2)

$(m \gg= f) \gg= g = m \gg= (\lambda x \rightarrow f x \gg= g)$

(L3)

Considereu aquestes dues funcions:

$rep1 :: \text{Int} \rightarrow \text{IO}()$

$rep1 0 io = \text{return}()$

$rep1 n io = \text{do}$

(11)

$io$

$rep1 (n - 1) io$

(12)

$rep2 :: \text{Int} \rightarrow \text{IO}() \rightarrow \text{IO}()$

$rep2 n io = \text{foldr} (\gg) (\text{return}()) (\text{replicate} n io)$

(2)

1. Digueu què fa  $rep1 4$  (**putStrLn** "Hola").

2. Demostreu que  $rep1 = rep2$ .

3. Demostreu que, per a tot parell de naturals  $p$  i  $q$ ,

$rep1 (p + q) io = rep1 p io \gg rep1 q io$

Estructureu correctament les vostres demostracions i anoteu-ne cada pas realitzat amb el codi de l'equació utilitzada (R1, R2, ..., HI, ...).

## 5 Tipus i subtipus en C++ i Java

(2 punts)

- Expliqueu què és un error de tipus, doneu la definició de llenguatge amb seguretat de tipus (*type-safety*) i digueu quan es pot comprobar la seguretat de tipus.
- Considereu aquest programa en C++ (que utilitza els vells arrays de C i no els nous vectors de C++), amb un únic error de compilació a la darrera línia:

```
struct Animal { };
struct Gat : public Animal { };
struct Gos : public Animal { };

void f (Animal* animals[]) {
    animals[0] = new Gos();
}

int main() {
    Gat* gats[] = { new Gat(), new Gat() };
    f(gats);
}
```

*← error: invalid conversion*

Tenint en compte l'error de compilació, expliqueu si el constructor d'arrays en C++ és covariant, contravariant o invariant.

- Considereu ara aquest programa en Java (essencialment idèntic a l'anterior) però que compila sense cap diagnòstic:

```
class Animal { }
class Gat extends Animal { }
class Gos extends Animal { }

class Programa {

    public static void f(Animal[] animals) {
        animals[0] = new Gos();
    }

    public static void main(String[] args) {
        Gat[] gats = new Gat[] { new Gat(), new Gat() };
        f(gats);
    }
}
```

Tenint en compte que el programa compila perfectament, expliqueu si el constructor d'arrays en Java és covariant, contravariant o invariant.

- Com pot ser que hom consideri que Java és un llenguatge amb seguretat de tipus quan la crida a *f(gats)* provocarà un error de tipus quan s'assigni un *Gos* en una llista de *Gats*? Altrament dit, com pot fer Java per evitar que es doni aquest error de tipus?

# Llenguatges de Programació, FIB, 21 de juny de 2023

## Possible solució

### 1 $\lambda$ -càlcul en Python amb una IA

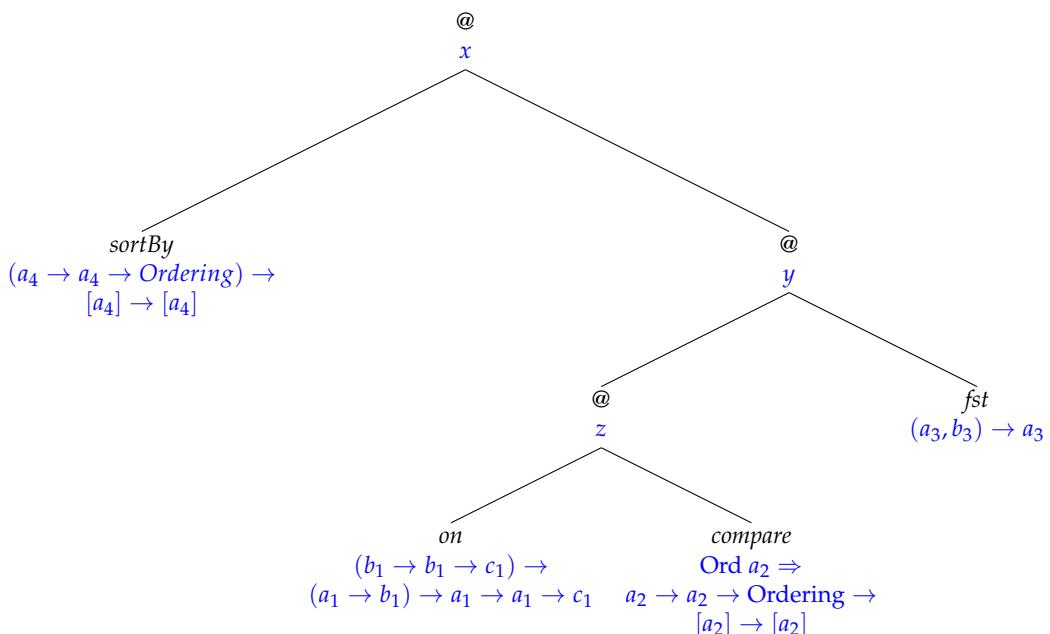
Les respostes de la IA són molt encertades i el codi generat és quasi perfecte, però aquest té un *bug*: Quan troba una aplicació que no té abstracció a l'esquerra, retorna una aplicació amb la reducció per l'esquerra i amb la reducció per la dreta (a la línia `App(normal_eval(lft), normal_eval(rgt))`). Però com que, en ordre normal, la reducció s'ha de fer el més amunt i a l'esquerra de l'arbre possible, si es fa a l'esquerra, ja no s'ha de fer a la dreta.

### 2 APL

1.  $\lceil$  calcula el màxim de dos valors  
/ insereix l'operador de l'esquerra entre cada element del vector de la dreta  
 $\approx$  inverteix l'ordre dels operands de l'operador de l'esquerra
2.  $\lceil$  és `max` en Haskell i `max` en Python  
/ és un `fold` en Haskell i `reduce` en Python  
 $\approx$  és un `flip` en Haskell i no té equivalent (predefinit) en Python
3.  $1 - \approx 2 \times a \lceil 3 \div b + c$
4.  $\times / \wr n$
5.  $(+/v) \div \not\equiv v$

### 3 Inferència de tipus

Aquest és l'arbre de sintaxi abstracta que correspon a *mySort* tot anotant els nodes amb el seu tipus quan és conegut i amb variables de tipus quan és desconegut.



L'arbre engendra aquestes restriccions que es poden anar solucionant de baix cap a amunt:

- $(b_1 \rightarrow b_1 \rightarrow c_1) \rightarrow (a_1 \rightarrow b_1) \rightarrow a_1 \rightarrow a_1 \rightarrow c_1 = (a_2 \rightarrow a_2 \rightarrow \text{Ordering} \rightarrow [a_2] \rightarrow [a_2]) \rightarrow z$   
 $\Rightarrow z = (a_1 \rightarrow b_1) \rightarrow a_1 \rightarrow a_1 \rightarrow c_1 \text{ i } b_1 = a_2 \text{ i } c_1 = \text{Ordering} \rightarrow [a_2] \rightarrow [a_2]$
- $(a_1 \rightarrow b_1) \rightarrow a_1 \rightarrow a_1 \rightarrow c_1 = ((a_3, b_3) \rightarrow a_3) \rightarrow y$   
 $\Rightarrow y = a_1 \rightarrow a_1 \rightarrow c_1 \text{ i } a_1 = (a_3, b_3) \text{ i } b_1 = a_3 = a_2.$
- $(a_4 \rightarrow a_4 \rightarrow \text{Ordering}) \rightarrow [a_4] \rightarrow [a_4] = (a_1 \rightarrow a_1 \rightarrow c_1) \rightarrow x$   
 $\Rightarrow x = [a_4] \rightarrow [a_4] \text{ i } a_4 = a_1 = (a_3, b_3).$

Fent  $a = a_3$  i  $b = b_3$ , queda  $\text{mySort} :: \text{Ord } a \Rightarrow [(a, b)] \rightarrow [(a, b)]$

## 4 Raonament equacional

1.  $\text{rep1 4 (putStrLn "Hola")}$  escriu quatre cops Hola i retorna un **IO ()**.

[Errors freqüents (no penalitzats): Dir que escriu "pel terminal"(escriu pel stdout, que pot ser un terminal, un fitxer, una impressora, un socket, ...). No dir què retorna.]

2. Primer desensucrem

$$\begin{aligned} \text{rep1 } n \text{ io} &= \text{do} \\ &\quad \text{io} \\ &\quad \text{rep1 } (n - 1) \text{ io} \end{aligned}$$

en

$$\text{rep1 } n \text{ io} = \text{io} \gg \text{rep1 } (n - 1) \text{ io} \quad -- = 12$$

Per demostrar que  $\text{rep1} = \text{rep2}$ , demostrarem que  $\text{rep1 } n \text{ io} = \text{rep2 } n \text{ io}$  per a qualsevol  $n$  (natural) i qualsevol  $\text{io}$  per inducció sobre  $n$ .

- Cas base:  $n = 0$ . Per una banda,

$$\text{rep1 } 0 \text{ io} = \text{return } () \quad -- 11$$

i, per altra banda,

$$\begin{aligned} \text{rep2 } 0 \text{ io} &= \text{foldr } (\gg) (\text{return } ()) (\text{replicate } 0 \text{ io}) \quad -- 2 \\ &= \text{foldr } (\gg) (\text{return } ()) [] \quad -- R1 \\ &= \text{return } () \quad -- F1 \end{aligned}$$

Per tant,  $\text{rep1 } 0 \text{ io} = \text{rep2 } 0 \text{ io}$ .

- Pas inductiu: Agafem com a hipòtesi d'inducció que  $\text{rep1 } n \text{ io} = \text{rep2 } n \text{ io}$ .  
 Per una banda,

$$\text{rep1 } (n+1) \text{ io} = \text{io} \gg \text{rep1 } n \text{ io} \quad -- 12$$

i, per altra banda,

$$\begin{aligned} \text{rep2 } (n+1) \text{ io} &= \text{foldr } (\gg) (\text{return } ()) (\text{replicate } (n+1) \text{ io}) \quad -- 2 \\ &= \text{foldr } (\gg) (\text{return } ()) (\text{io} : \text{replicate } n \text{ io}) \quad -- R2 \\ &= \text{io} \gg \text{foldr } (\gg) (\text{return } ()) (\text{replicate } n \text{ io}) \quad -- F2 \\ &= \text{io} \gg \text{rep2 } n \text{ io} \quad -- 2 \\ &= \text{io} \gg \text{rep1 } n \text{ io} \quad -- HI \end{aligned}$$

Per tant,  $\text{rep1 } (n+1) \text{ io} = \text{rep2 } (n+1) \text{ io}$ .

- I queda doncs demostrat que  $\text{rep1} = \text{rep2}$ .

3. Primer demostrem aquest lema (A) sobre l'associativitat de  $\gg$ :

$$\begin{aligned} m \gg (f \gg g) &= m \gg= (\lambda_1 \rightarrow f \gg= (\lambda_1 \rightarrow g)) && \text{-- D} \\ &= (m \gg= (\lambda_1 \rightarrow f)) \gg= (\lambda_1 \rightarrow g) && \text{-- L3} \\ &= (m \gg f) \gg g && \text{-- D} \end{aligned}$$

Ara demostrem que, per a tot parell de naturals  $p$  i  $q$ ,

$$\text{rep1 } (p + q) \text{ io} = \text{rep1 } p \text{ io} \gg \text{rep1 } q \text{ io}$$

Ho fem per inducció sobre  $p$  (amb  $q$  qualsevol):

- Cas base:  $p = 0$ . Tenim

$$\begin{aligned} \text{rep1 } 0 \text{ io} \gg \text{rep1 } q \text{ io} &= \\ &= \text{return } () \gg \text{rep1 } q \text{ io} && \text{-- 11} \\ &= \text{return } () \gg= (\lambda_1 \rightarrow \text{rep1 } q \text{ io}) && \text{-- D} \\ &= (\lambda_1 \rightarrow \text{rep1 } q \text{ io}) () && \text{-- L1} \\ &= \text{rep1 } q \text{ io} && \text{-- aplicació} \\ &= \text{rep1 } (0+q) \text{ io} && \text{-- element neutre} \end{aligned}$$

- Pas inductiu: Agafem  $\text{rep1 } (p+q) \text{ io} = \text{rep1 } p \text{ io} \gg \text{rep1 } q \text{ io}$  com a hipòtesi d'inducció. Llavors,

$$\begin{aligned} \text{rep1 } (1+p+q) \text{ io} &= \\ &= \text{io} \gg \text{rep1 } (p+q) \text{ io} && \text{-- 12} \\ &= \text{io} \gg (\text{rep1 } p \text{ io} \gg \text{rep1 } q \text{ io}) && \text{-- HI} \\ &= (\text{io} \gg \text{rep1 } p \text{ io}) \gg \text{rep1 } q \text{ io} && \text{-- A} \\ &= \text{rep1 } (1+p) \text{ io} \gg \text{rep1 } q \text{ io} && \text{-- 12} \end{aligned}$$

- I queda doncs demostrada la propietat.

[Errors freqüents: Utilitzar punts suspensius. Ignorar o doanr per feta l'associativitat de l'operador  $\gg$ .]

## 5 Tipus i subtipus en C++ i Java

1. Un error de tipus consisteix en aplicar a les dades una operació que el seu tipus no suporta. La seguretat de tipus (*type safety*) és la mesura de com de fàcil/difícil és cometre errors de tipus en un LP. La comprovació de tipus pot ser estàtica (en temps de compilació), dinàmica (en temps d'execució) o mixta (ambdós).
2. En C++ tenim doncs que  $Gat \leq Animal$  però que  $\text{array}[Gat] \not\leq \text{array}[Animal]$ . Per tant, el constructor d'arrays en C++ no és covariant. Com que no té gaire sentit que sigui contravariant, és invariant.
3. En Java, com que el programa compila,  $\text{array}[Gat] \leq \text{array}[Animal]$ . Per tant, el constructor d'arrays en Java és covariant.
4. Com que els arrays de Java són covariants, en temps de compilació no hi ha cap problema en passar un array de *Gats* a una funció que espera un array d'*Animals*. Però quan es fa, la línia `animals[0] = new Gos();` introduceix un error de tipus perquè, malgrat que el tipus d'*animals* sigui array d'*Animals* en temps de compilació,

en temps d'execució és un array de *Gats*. Ara bé, Java té un sistema de comprovació de tipus mixt i, en temps d'execució, aquesta assignació lanza una excepció d'error de tipus.

# Llenguatges de Programació, FIB, 13 de gener de 2023

*L'examen dura tres hores. Poseu el vostre nom a cada full. Contesteu cada problema en fulls diferents. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de les respostes. Feu bona lletra.*

## 1 Lambda càcul (1 punt)

Haskell Curry va descobrir que el combinador Y definit per

$$Y = \lambda z.(\lambda x.z(xx))(\lambda x.z(xx))$$

satisfà la propietat de punt fixe:  $Y R = R(Y R)$ .

Més tard, Alan Turing va descobrir un altre combinador de punt fixe anomenat  $\Theta$ :

$$\Theta = (\lambda xy.y(xxy)) (\lambda xy.y(xxy))$$

Demostreu que  $\Theta$  és, en efecte, un altre combinador de punt fixe.

## 2 Compilació (1 punt)

Escriviu en ANTLR una gramàtica per expressions en lambda càcul y discutiu si la solució presentada és una gramàtica LL(1).

## 3 Lleis de functors (2 punts)

Recordeu que la classe **Functor** ve definida per

```
class Functor f where
    fmap :: (a → b) → (f a → f b)
```

i que les lleis dels functors són:

1.  $fmap(g1 . g2) = fmap g1 . fmap g2$
2.  $fmap \text{id} = \text{id}$

Considereu aquesta definició per arbres binaris i la seva instanciació com a functor:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

```
instance Functor BinTree where
    fmap f Empty = Empty
    fmap f (Node v t1 t2) = Node (f v) (fmap f t1) (fmap f t2)
```

Comproveu si aquesta instanciació compleix cadascuna de les lleis dels functors.

## 4 La funció join

(2 punts)

Recordeu la definició de la classe **Monad**:

```
class Monad m where
    return :: a → m a
    (≫=) :: m a → (a → m b) → m b
```

Recordeu també la instanciació de **Maybe** i llistes com a mònades:

```
instance Monad Maybe where
    return      =  Just
    Nothing ≫= f =  Nothing
    Just x ≫= f =  f x

instance Monad [] where
    return x      =  [x]
    xs ≫= f       =  concat (map f xs)
```

El paquet Control.Monad ofereix una funció **join** definida d'aquesta forma:

```
join k = do
    m ← k
    m
```

1. Calculeu (utilitzant reescritura i deixant clar cada pas) quan valen les expressions següents:
  - (a) **join Nothing**
  - (b) **join (Just (Just 3))**
  - (c) **join (Just Nothing)**
  - (d) **join [[1, 2], [3, 4]]**
2. Inferiu el tipus complet de **join**. Per a fer-ho, dibuixe l'arbre de sintaxi abstracta corresponent a **join**, anoteu l'arbre amb els tipus de cada node, genereu el conjunt de restriccions d'igualtat de tipus i resoleu les restriccions. No us deixeu els requeriments sobre les classes.

## 5 Python amb tipus

(2 punts)

Malgrat que Python és un llenguatge de tipus dinàmic, des de fa un temps ofereix la possibilitat d'anotar els tipus de les variables, paràmetres i funcions. En temps d'execució, aquestes anotacions no provoquen cap impacte en la manera que s'executen els programes ni sobre el seu temps d'execució: simplement es consideren com a comentaris. L'objectiu d'aquestes anotacions de tipus és permetre que altres eines, com ara `mypy`, puguin verificar estàticament l'absència d'error de tipus.

Per exemple, en el següent programa en Python amb tipus, `mypy` diagnostica que la crida a factorial és incorrecta abans d'executar el programa (estalviant així un error de tipus en temps d'execució):

```
def factorial (n: int) → int: ...
print(factorial ([1,2,3]))
```

En Python, el tipus de les llistes d'elements de tipus  $T$  s'escriu `list [T]`. Per exemple, aquesta seria la capçalera d'una funció que retornaria la mitjana dels elements en una llista d'enters:

```
def mitjana(L: list [int]) → float: ...
```

El tipus de les tuples de  $T_1, T_2, \dots$  s'escriu `tuple[T1, T2, ...]`. Per exemple, aquesta seria la capçalera d'una funció que retornaria la distància entre dos punts:

```
def distancia (p: tuple[float, float], q: tuple[float, float]) → float: ...
```

Considereu aquest fragment de programa:

```
class Animal: ...
class Gos(Animal): ...
```

```
parella_animal : tuple[Animal, Animal] = ...
parella_gos : tuple[Gos, Animal] = ...
```

Llavors, si s'escriu

```
parella_gos = parella_animal
```

`mypy` dóna un error, però si s'escriu

```
parella_animal = parella_gos
```

`mypy` no dóna cap error.

En canvi, per aquest fragment de programa

```
class Animal: ...
class Gos(Animal): ...
```

```
llista_animals : list [Animal] = ...
llista_gossos : list [Gos] = ...
```

si s'escriu

```
llista_gossos = llista_animals
```

`mypy` dóna un error, i si s'escriu

```
llista_animals = llista_gossos
```

`mypy` també dóna un error.

Utilitzant la informació anterior, contesteu les preguntes següents:

1. Doneu les definicions de constructor de tipus covariant, contravariant i invariant.
2. Digueu si per a `mypy`, el constructor de tipus `tuple` és covariant, contravariant i/o invariant i expliqueu perquè.
3. Digueu si per a `mypy`, el constructor de tipus `list` és covariant, contravariant i/o invariant i expliqueu perquè.
4. Expliqueu per quina raó té sentit que la variància de `tuple` i `list` sigui diferent.

## 6 Subtipus

(2 punts)

Considereu les declaracions següents en Java (tot ignorant qüestions de visibilitat):

```
class Empleat {
    int code, m;

    Empleat f (Empleat x) {
        if (code == x.code) return x; else return this;
    }

    void h (int x) { m = x; }

}

class Venedor extends Empleat {
    int n;

    Venedor f (Venedor x) {
        if (code == x.code) return this; else return x;
    }

    void g (int x) { m = x; }

}

class Comptable extends Empleat {
    int a;

    Comptable f (Comptable x) {
        if (code == x.code) return x; else return this;
    }

    void h (int x) { a = x; }

}
```

Donades les inicialitzacions següents:

```
Comptable c = new Comptable(); c.code = 10;
Venedor v = new Venedor();      v.code = 10;
Empleat e = new Empleat();      e.code = 10;
```

i donades les seqüències d'instruccions següents:

- (a)  $e = e.f(v); \quad e.g(7);$
- (b)  $v = e.f(v); \quad v.g(7);$
- (c)  $e = c.f(c); \quad e.h(7);$
- (d)  $e = e.f(c); \quad e.h(7);$

1. Digueu els errors de tipus que contenen indicant, a més a quina funció  $f$  i  $h$  es crida en cada cas correcte.
2. Escriviu, en C++, una classe semànticament equivalent a la classe *Empleat* de Java.

# Llenguatges de Programació, FIB, 13 de gener de 2023

## Possibles solucions i errors freqüents

### 1 Lambda càcul

Definint  $F = \lambda xy . y(xxy)$  tenim que  $\Theta = FF$ . Llavors,

$$\begin{aligned}\Theta R &= (FF)R \\ &= ((\lambda xy . y(xxy))F)R \\ &= (\lambda y . y(FFy))R \\ &= R(FFR) \\ &= R(\Theta R)\end{aligned}$$

tal com cal.

### 2 Compilació

```
grammar lambda;

expressio   : VARIABLE
            | '(' expressio ')'
            | expressio expressio
            | LAMBDA VARIABLE '.' expressio
            ;
VARIABLE    : [a-zA-Z] ;
LAMBDA      : 'L' ;
WS          : [ \t\n\r]+ -> skip ;
```

### 3 Lleis de functors

(Usem  $E$  per *Empty* i  $N$  per *Node*.)

1. Primera llei dels functors: Comprovem que es compleix fent un anàlisi per casos:

- Per una banda,

$$fmap(g1 . g2) E = E$$

i per altra banda

$$\begin{aligned}(fmap g1 . fmap g2) E &= (fmap g1) ((fmap g2) E) \\ &= (fmap g1) E \\ &= E\end{aligned}$$

- Per una banda,

$$fmap(g1 . g2) (N v t1 t2) = N((g1.g2)v) E E$$

i per altra banda

$$\begin{aligned}
& (fmap g1 \cdot fmap g2) (N v t1 t2) \\
&= (fmap g1) ((fmap g2) (N v t1 t2)) \\
&= (fmap g1) (N (g2 v) E E) \\
&= N (g1 (g2 v)) E E \\
&= N ((g1 \cdot g2) v) E E
\end{aligned}$$

2. Segona llei dels functors: No es compleix: Per exemple,

$$fmap \mathbf{id} (N 2 (N 1 E E) E) = N 2 E E$$

i

$$\mathbf{id} (N 2 (N 1 E E) E) = N 2 (N 1 E E) E$$

però  $N 2 E E \neq N 2 (N 1 E E) E$ .

## 4 La funció join

(Usem  $N$  per **Nothing** i  $J$  per **Just**.)

Tenim que

$$\begin{aligned}
\mathbf{join} k &= \mathbf{do} \\
&\quad m \leftarrow k \\
&\quad m
\end{aligned}$$

és sucre sintàctic per a

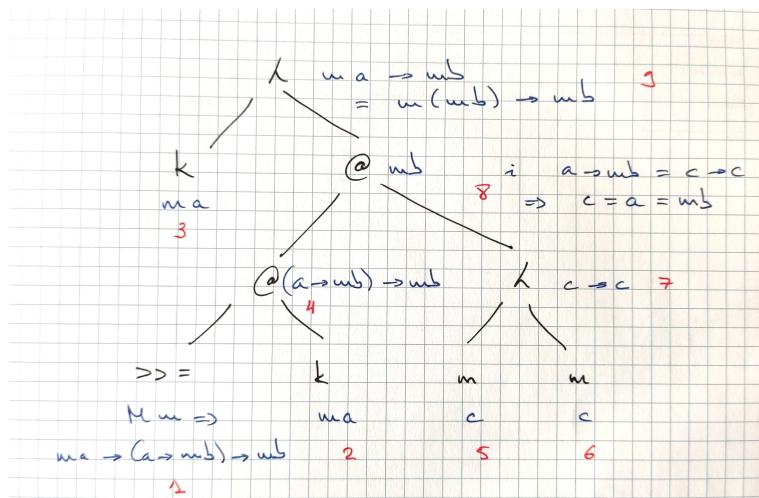
$$\mathbf{join} k = k \gg= (\lambda m \rightarrow m)$$

que equival a

$$\mathbf{join} k = k \gg= \mathbf{id}$$

1. (a)  $\mathbf{join} N = N \gg= \mathbf{id} = N$
- (b)  $\mathbf{join} (J (J 3)) = (J (J 3)) \gg= \mathbf{id} = \mathbf{id} (J 3) = J 3$
- (c)  $\mathbf{join} (J N) = (J N) \gg= \mathbf{id} = \mathbf{id} N = N$
- (d)  $\mathbf{join} [[1,2],[3,4]] = [[1,2],[3,4]] \gg= \mathbf{id} = \mathbf{concat} (\mathbf{map} \mathbf{id} [[1,2],[3,4]])$   
 $= \mathbf{concat} ([[1,2],[3,4]]) = [1,2,3,4]$

2. El tipus de **join** és **Monad**  $m \Rightarrow m (m b) \rightarrow m b$ . L'esquema següent mostra els passos realitzats.



## 5 Python amb tipus

1. Sigui  $C$  un constructor de tipus i siguin  $s$  i  $t$  dos tipus tals que  $s \leq t$ . Entenem que  $s \leq t$  si en tot context on es pot usar un objecte de tipus  $t$ , es pot usar un objecte de tipus  $s$ .  
Si  $C[s] \leq C[t]$  llavors  $C$  es covariant.  
Si  $C[t] \leq C[s]$  llavors  $C$  es contravariant.  
Si no es té  $C[s] \leq C[t]$  ni  $C[t] \leq C[s]$  llavors  $C$  es invariant.
2. Per la relació d'herència, tenim que  $Gos \leq Animal$ . Com que l'assignació  $parella_animal = parella_gos$  és vàlida, vol dir que  $\text{tuple}[Gos, ...] \leq \text{tuple}[Animal, ...]$ . Per tant, **tuple** és covariant.
3. Per la relació d'herència, tenim que  $Gos \leq Animal$ . Com que les assignacions  $llista_animals = llista_gossos$  i  $llista_gossos = llista_animals$  són invàlides, no es té  $llista[Gos] \leq \text{list}[Animal]$  ni  $\text{list}[Animal] \leq \text{list}[Gos]$ . En conseqüència, **list** és invariant.
4. En Python, les llistes són mutables mentre que les tuples no ho són. Si les llistes fossin covarians com les tuples els elements de les llistes podrien “escapar-se” del seu tipus. Per exemple, en el codi següent

```
class Animal: ...
class Gos(Animal): ...
class Gat(Animal): ...

def afegir_gat(animals: list[Animal]) → None:
    animals.append(Gat())
    # cap problema!

gossos: list[Gos] = ...
afegir_gat(gossos)
# ai ai!
```

es podria posar un *Gat* en una llista de *Gossos* a través d'*afegir\_gat*.

## 6 Subtipus

1. (a) La crida  $e.g(7)$ ; no és correcta ja que  $e$  té tipus estàtic *Empleat* i la classe *Empleat* no té el mètode  $g$ . Es crida a la funció  $f$  d'*Empleat*.  
(b) L'assignació no és correcta ja que  $v$  té tipus *Venedor* i  $e.f(v)$  té tipus *Empleat*. Es crida a la funció  $f$  d'*Empleat*.  
(c) Les dues instruccions són correctes. Es crida a les funcions  $f$  i  $h$  de *Contable*.  
(d) Les dues instruccions són correctes. Es crida a les funcions  $f$  i  $h$  de *Contable*.
2. Es pot fer amb referències:

```
class Empleat {
    int code, m;

    virtual Empleat& f(Empleat& x) {
        if (code == x.code) return x; else return *this;
    }
}
```

```
    virtual void h (int x) { m = x; }  
}
```

O amb punters:

```
class Empleat {  
    int code, m;  
  
    virtual Empleat* f (Empleat* x) {  
        if (code == x.code) return x; else return this;  
    }  
  
    virtual void h (int x) { m = x; }  
}
```

# Llenguatges de Programació, FIB, 10 de juny de 2022

*L'examen dura tres hores. Feu bona lletra. Poseu el vostre nom a cada full. Contesteu cada problema enfulls diferents. Cada problema compta 2'5 punts. Es valorarà la concisió, claredat i brevetat a més de la completeness i l'exactitud de les respostes.*

## 1 Inferència de tipus i el combinador Y

**Observació:** En aquest exercici haureu d'inferir els tipus d'algunes expressions. Per a fer-ho, dibuixeu l'arbre de sintàxi abstracta de l'expressió, anoteu l'arbre amb els tipus de cada node, genereu el conjunt de restriccions d'igualtat de tipus i resoleu les restriccions.

Recordeu que en el  $\lambda$ -càlcul, el combinador Y definit per

$$Y = \lambda z.(\lambda x.z(xx))(\lambda x.z(xx))$$

satisfà que  $Y R = R(Y R)$ .

En Haskell, es podria provar doncs d'escriure així:

$$\text{combinador}Y z = (\lambda x \rightarrow z(x\,x)) (\lambda x \rightarrow z(x\,x))$$

- (1) Mostreu que *combinadorY* no tipa (és a dir, que conté un error de tipus).

Considereu aquesta alternativa:

$$\begin{aligned} \text{data } Ma &= M(M\,a \rightarrow a) && \text{-- } M \text{ és alhora el nom del tipus i del constructor} \\ && & \text{-- i no té res a veure amb cap mònada} \\ w\,h &= h\,(M\,h) \end{aligned}$$

$$y\,f = w\,(\lambda(M\,x) \rightarrow f(w\,x))$$

- (2) Inferiu el tipus de  $w$ .  
(3) Inferiu el tipus de  $y$ .  
(4) Demostreu que  $y\,r = r\,(y\,r)$ .

## 2 Llistes i mònades

Recordeu que la classe Monad compta amb aquestes operacions:

```
class Monad m where
  (>>) :: m a → (a → m b) → m b
  return :: a → m a
```

Les llistes instancien les mònades d'aquesta forma:

```
instance Monad [] where
  xs `>` f = concat (map f xs)
  return x = [x]
```

- (1) Sigui  $f x = [x, 2*x]$  i  $xs = [1, 2, 3]$ . Calculeu quan val  $xs \gg= f$  deixant clar cada pas efectuat.

La sintàxi de les llistes per comprensió no és altra cosa que sucre sintàctic. En efecte, l'expressió  $[f x y \mid x \leftarrow xs, y \leftarrow ys]$  és equivalent a

**do**

```
x ← xs
y ← ys
return (f x y)
```

- (2) Desensucreu totalment (és a dir, escriviu en notació funcional sense sucre sintàctic i usant les funcions/operadors de **Monad**) l'expressió  $[(x,y) \mid x \leftarrow "abc", y \leftarrow [1, 2, 3]]$ .

Per poder disposar de filtres en les llistes de comprensió, les mònades de Haskell també tenen una operació **guard** [la implementació real és més complicada que això però aquesta descripció és suficient per aquest exercici]:

```
class Monad m where
  (≫=) :: m a → (a → m b) → m b
  return :: a → m a
  guard :: Bool → m ()
```

Llavors, els filtres en les llistes de comprensió són sucre sintàctic sobre la funció **guard**. Per exemple, l'expressió  $[f x y \mid x \leftarrow xs, p x, y \leftarrow ys]$  és equivalent a

**do**

```
x ← xs
guard (p x)
y ← ys
return (f x y)
```

- (3) Desensucreu totalment (és a dir, escriviu en notació funcional sense sucre sintàctic i usant les funcions/operadors de **Monad**) l'expressió següent:

$[(x, y, z) \mid x \leftarrow [1..n], y \leftarrow [x..n], z \leftarrow [y..n], x*x + y*y == z*z]$

- (4) Implementeu **guard** per a llistes, de forma que les llistes per comprensió funcionin correctament amb filtres.

- (5) Com caldria desensucrar els **let** de les llistes per comprensió? Expliqueu-ho aplicant-ho a l'expressió següent:

```
[(x, y, z) |
  x ← [1..n],
  y ← [x..n],
  let z = isqrt (x*x + y*y),
  z ≤ n,
  x*x + y*y == z*z
]
```

### 3 Raonament equacional

Considerant les definicions següents,

$$\begin{aligned} s [] &= 0 & (a_1) \\ s (x:xs) &= x + s \ xs & (a_2) \end{aligned}$$

$$\begin{aligned} \ell [] &= 0 & (b_1) \\ \ell (x:xs) &= 1 + \ell xs & (b_2) \end{aligned}$$

$$\begin{aligned} c [] &= [] & (c_1) \\ c (x:xs) &= x ++ c xs & (c_2) \end{aligned}$$

$$\begin{aligned} m\_ [] &= [] & (d_1) \\ mf (x:xs) &= f x : mf xs & (d_2) \end{aligned}$$

$$(f . g) x = f (g x) \quad (e)$$

demostreu per inducció que

$$\ell . c = s . (m \ \ell)$$

Si us cal, podeu utilitzar aquest lema:

$$\ell (x ++ y) = \ell x + \ell y \quad (L)$$

Anoteu cada pas amb l'etiqueta de l'equació que heu utilitzat. Utilitzeu HI per referir-vos a l'ús de l'hipòtesi d'inducció.

### 4 Subtipos

**Nota:** En este problema se supone que todas las declaraciones son public.

Dadas las siguientes declaraciones en C++:

```
class punto {
    double x,y;
    void mover1(double x1, double y1) {x += x1; y += y1;}
    void reset() {x = 0.0; y = 0.0;}
};

class puntocolor: punto {
    int color;
    void cambiar_color(int n) {color = n;}
    void reset() {x = 0.0; y = 0.0; color = 0;}
};

punto mover2 (punto p, double x1, double y1) {
    p.x += x1; p.y += y1;
    return p;
}
```

y dadas las declaraciones similares en Java:

```

class punto {
    double x,y;
    void mover1(double x1, double y1) { x += x1; y += y1; }
    void reset() { x = 0.0; y = 0.0; }
}

class puntocolor extends punto {
    int color;
    void cambiar_color(int n) { color = n; }
    void reset() { x = 0.0; y = 0.0; color = 0; }
}

punto mover2 (punto p, double x1, double y1) {
    p.x += x1; p.y += y1;
    return p;
}

```

Contestad las siguientes preguntas, justificando las respuestas:

1. Dada la declaración en C++ punto\* p; ¿qué tipo tendría p1 en el siguiente fragmento de código? y ¿A qué versión de reset se llamaría en el siguiente fragmento de código?

```

p = new puntocolor;
...
auto p1 = mover2(*p,2.0,1.0);
p1.reset();

```

2. Dada la declaración punto p; puntocolor c; ¿Sería correcto en C++ o en Java el siguiente código?

```

c = mover2(c,2.0,1.0);
c.cambiar_color(0);

```

3. Dada la declaración punto p; puntocolor c; ¿Sería correcto en C++ o en Java el siguiente fragmento de código?

```

p = mover2(c,2.0,1.0);
p.cambiar_color(0);

```

4. Con las mismas declaraciones que en el caso anterior, ¿Sería correcto en C++ o en Java el siguiente código? y ¿a qué versión de reset se llamaría en esos lenguajes en el siguiente fragmento de código?

```

c.mover1(2.0,1.0);
c.cambiar_color(0);
p = c;
p.reset();

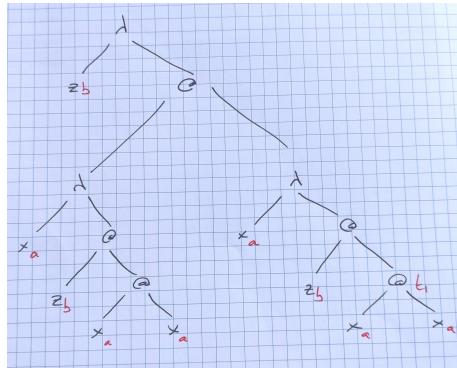
```

# Llenguatges de Programació, FIB, 10 de juny de 2022

## Possibles solucions i errors freqüents

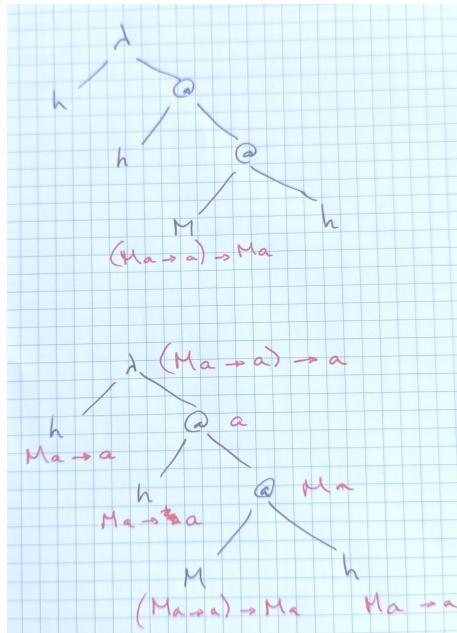
### 1 Inferència de tipus i el combinador Y

1. L'arbre de sintàxi anotat amb tipus és el següent:

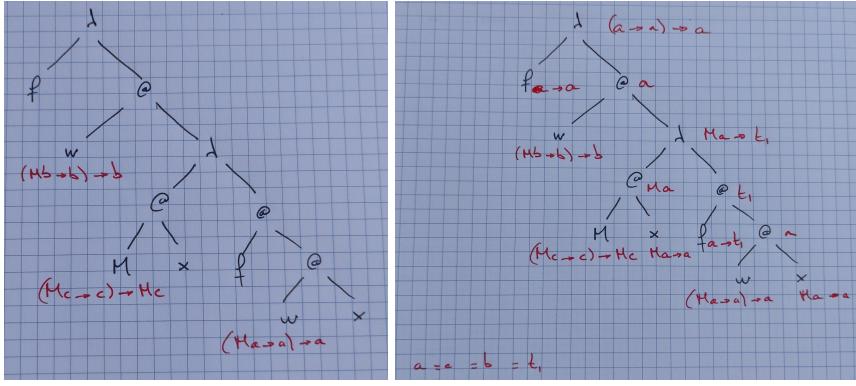


Només escriure la restricció del node de més a la dreta ens trobem un problema: La restricció és  $a = a \rightarrow t_1$  que correspon a crear un tipus infinit.

2. L'arbre de sintàxi és a la part superior de la figura. Només cal etiquetar el tipus de  $M$  correctament i anar pujant per l'arbre per trobar que  $w :: (M a \rightarrow a) \rightarrow a$  i tots els tipus de les subexpressions (part inferior de la figura).



3. L'arbre de sintàxi és a la part esquerra de la figura, amb les anotacions dels tipus coneguts. Després es poden anar inferint tota la resta de tipus de dalt a baix (part dreta de la figura) fins trobar que  $y :: (a \rightarrow a) \rightarrow a$ .



4. Tenim:

$$\begin{aligned}
 y \ r &= \\
 &= w (\lambda(M x) \rightarrow r(w x)) && \text{-- def de } y \\
 &= (\lambda(M x) \rightarrow r(w x)) (M (\lambda(M x) \rightarrow r(w x))) && \text{-- def de } w \\
 &= (\lambda(M x) \rightarrow r(w x)) (M (\lambda(M z) \rightarrow r(w z))) && \text{-- renombrament} \\
 &= r (w (\lambda(M z) \rightarrow r(w z))) && \text{-- aplicació} \\
 &= r (w (\lambda(M x) \rightarrow r(w x))) && \text{-- renombrament} \\
 &= r (y \ r) && \text{-- primera equació}
 \end{aligned}$$

I per tant hem implementant un punt fixe en Haskell sense errors de tipus.

### Errors freqüents

- Interpretar  $z(x \ x)$  como  $(z \ x)x$ .
- Interpretar que  $x \ x$  es una única variable y no  $x$  aplicado a  $x$ .
- No saber reconocer el error de tipado.
- No saber definir el tipo de  $M$ .
- (1.4) Pensar que lo que se pedía era ver que el tipo de  $y \ r$  es el mismo que el tipo de  $r(y \ r)$ .
- (1.4) Pensar que se pedía ver  $Y F = F (Y F)$ .

## 2 Llistes i mònades

1.  $xs \gg= f =$ 

$$\begin{aligned}
 &= \text{concat} (\text{map} \ f \ xs) && \text{-- definició de } \gg= \\
 &= \text{concat} (\text{map} (\lambda x \rightarrow [x, 2*x]) [1, 2, 3]) && \text{-- definició de } f \text{ i } xs \\
 &= \text{concat} ([[1, 2], [2, 4], [3, 6]]) && \text{-- aplicació de map} \\
 &= [1, 2, 2, 4, 3, 6] && \text{-- aplicació de concat}
 \end{aligned}$$

### 2. La llista per comprensió

$$[(x,y) \mid x \leftarrow "abc", y \leftarrow [1, 2, 3]]$$

és desensucra a notació **do**:

```

do
  x ← "abc"
  y ← [1, 2, 3]
  return (x,y)
  
```

i aquesta es desensucra a notació funcional:

```
"abc" >>= \x →
[1, 2, 3] >>= \y →
return (x,y)
```

### 3. La llista per comprensió

$[(x,y,z) \mid x \leftarrow [1..n], y \leftarrow [x..n], z \leftarrow [y..n], x*x + y*y == z*z]$

és desensucra a notació **do**:

```
do
  x ← [1..n]
  y ← [x..n]
  z ← [y..n]
  guard (x*x + y*y == z*z)
  return (x,y,z)
```

i aquesta es desensucra a notació funcional:

```
[1.. n] >>= \x →
[x .. n] >>= \y →
[y .. n] >>= \z →
guard (x*x + y*y == z*z) >>
return (x,y,z)
```

### 4. Es pot fer així:

```
guard True = return ()
guard False = []
```

### 5. Per desensucrar els **lets** dins de llistes per comprensió, només cal posar-los dins la notació **do**, que ja té **lets**. En el cas de l'exemple quedaría així:

```
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)
```

## Errors freqüents

- Fer que **guard** no segueixi la seva signatura.
- Confondre **guard** i **filter**.
- No desensucrar *totalment*.
- Intentar avaluar expressions (menys la primera).
- Nota: No s'ha penalitzat usar  $\gg=$  enlloc de  $\gg$  després del **guard**, ja que  $\gg$  no apareixia a l'enunciat. En canvi, usar  $\rightarrow$  no té sentit.

## 3 Raonament equacional

- Cas base: la llista buida. Per una banda tenim:

$$\begin{aligned} (\ell . c) [] &= \\ &= \ell(c []) &-- (e) \\ &= \ell [] &-- (c_1) \\ &= 0 &-- (b_1) \end{aligned}$$

Per altra banda tenim:

$$\begin{aligned}
 (s . (m \ell)) [] &= \\
 &= s (m \ell []) \quad \text{-- (e)} \\
 &= s [] \quad \text{-- (d}_1\text{)} \\
 &= 0 \quad \text{-- (a}_1\text{)}
 \end{aligned}$$

Per tant,  $(\ell . c) [] = (s . (m \ell)) []$ .

- Cas inductiu: agafem

$$(\ell . c) xs = (s . (m \ell)) xs$$

com a hipòtesi d'inducció. Volem veure que  $(\ell . c) (x:xs) = (s . (m \ell)) (x:xs)$ :

Per una banda tenim:

$$\begin{aligned}
 (\ell . c) (x:xs) &= \\
 &= \ell (c (x:xs)) \quad \text{-- (e)} \\
 &= \ell (x ++ c xs) \quad \text{-- (c}_2\text{)} \\
 &= \ell x + \ell (c xs) \quad \text{-- (L)} \\
 &= \ell x + (\ell . c) xs \quad \text{-- (e)} \\
 &= \ell x + (s . (m \ell)) xs \quad \text{-- (HI)}
 \end{aligned}$$

Per altra banda tenim:

$$\begin{aligned}
 (s . (m \ell)) (x:xs) &= \\
 &= s (m \ell (x:xs)) \quad \text{-- (e)} \\
 &= s (\ell x : m \ell xs) \quad \text{-- (d}_2\text{)} \\
 &= \ell x + s (m \ell xs) \quad \text{-- (a}_2\text{)} \\
 &= \ell x + (s . (m \ell)) xs \quad \text{-- (e)}
 \end{aligned}$$

Per tant,  $(\ell . c) (x:xs) = (s . (m \ell)) (x:xs)$ .

Conseqüentment,  $\ell . c = s . (m \ell)$ .

## Errors freqüents

- No estructurar la demostració.
- No anunciar la hipòtesi d'inducció.
- No usar la hipòtesi d'inducció.
- Usar `[]` com a cas base.
- No anotar cada pas amb l'etiqueta de l'equació utilitzada.
- Utilitzar símbols direccionals per les equacions ( $\rightarrow$ ,  $\Rightarrow$  i semblants). Són igualtats.
- No seguir la sintàxi de Haskell.
- Usar el·lipsi (punts suspensius) en una demostració per inducció.

## 4 Subtipos

1. De acuerdo con la definición `punto mover2 (punto p, double x1, double y1)`, la llamada `mover2(*p, 2.0, 1.0)` es correcta, ya que `*p` tiene tipo estático `punto` y tipo dinámico `puntocolor`. También por su definición sabemos que el resultado de esa llamada tiene tipo `punto`. Por tanto, `p1` tiene tipo `punto`. Como consecuencia, la llamada `p1.reset()` será una llamada a la operación `reset` de la clase `punto`.
2. Sería incorrecto, tanto en C++ como en Java, ya que el resultado de `mover2` tiene tipo `punto` y no se puede asignar una expresión de tipo `punto` a una variable de tipo `puntocolor`.

3. Sería incorrecto, tanto en C++ como en Java, ya que la llamada `p.cambiar_color(0)` sería incorrecta, porque `p` es de tipo punto y `cambiar_color` está definida en la clase `puntocolor`.
4. Es correcto, tanto en C++ como en Java. En el caso de C++ la llamada `p.reset()` sería una llamada a la operación de la clase punto, ya que `p` tiene ese tipo estático y es un objeto estático. En cambio, en el caso de Java, la llamada sería a la operación de la clase `puntocolor`, ya que `p` tiene tipo dinámico `puntocolor`.

### Errors freqüents

- Pensar que `mover2(c,-,-)` es de tipo `puntocolor`, cuando `c` es de tipo `puntocolor`.
- Poner que en C++ o en Java no se permite `p = c;` cuando `p` es de tipo punto y `c` es de tipo `puntocolor`.
- Poner que en C++ o en Java se permite `c = p;` cuando `p` es de tipo punto y `c` es de tipo `puntocolor`.

# Llenguatges de Programació, FIB, 17 de gener de 2022

*L'examen dura tres hores. Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes. Feu bona lletra. Poseu el vostre nom a cada full. Contesteu els problemes 1 a 4 en fulls diferents dels problemes 5 a 7.*

## 1 Pragmàtica dels LPs (1 punt)

Perquè els llenguatges de la família del C (com C, C++, Java, ...) requereixen parèntesis al voltant de les condicions dels **ifs** i **whiles**?

Limiteu la vostra resposta a cinc línies o menys.

## 2 $\lambda$ -cálculo (1 punt)

Sabiendo que en el  $\lambda$ -cálculo **true** se representa por  $\lambda xy . x$  y **false** por  $\lambda xy . y$ , se pide demostrar que  $\lambda ab . ab(\lambda xy . y)$  es la implementación de la función **and**.

## 3 Subtipos (1 punt)

Explica si los punteros se pueden considerar covariantes, contravariantes o invariantes, justificando la respuesta. Es decir, que suponiendo que  $t_1$  es subtipo de  $t_2$ , si se puede considerar que  $t_1^*$  ha de ser subtipo de  $t_2^*$  o si, por el contrario,  $t_2^*$  ha de ser subtipo de  $t_1^*$  o si no se puede considerar ninguna de las dos cosas.

## 4 Inferència de tipus (2 punts)

Aquest *one-liner* trobat per Internet calcula el producte cartesià de dues llistes:

```
cartesian :: [a] → [a] → [[a]]  
cartesian = ((.) sequence) . (λx y → [x, y])
```

La definició sembla funcionar (per exemple, *cartesian* [1,2,3] [1,2] retorna el que cal: [[1,1], [1,2], [2,1], [2,2], [3,1], [3,2]]) però, malauradament, no sabem què és la funció *sequence*.

Inferiu el tipus de *sequence* seguint els passos següents:

1. Dibuixeu l'arbre de sintàxi abstracta de la definició de *cartesian* .
2. Anoteu l'arbre amb els tipus de cada node.
3. Genereu el conjunt de restriccions d'igualtat de tipus.
4. Resoleu les restriccions per trobar el tipus de *sequence*.

## 5 Python

(1 punt)

Implementeu en Python els decoradors *make\_bold*, *make\_italic*, i *make\_underline* que permetin formatejar en negreta, itàlica o subratllat d'HTML els textos retornats per altres funcions. Per exemple, amb

```
@make_bold
@make_italic
@make_underline
def foo():
    return "foo"

@make_italic
def bar():
    return "bar"
```

la crida a *foo()* hauria de retornar "<b><i><u>foo</u></i></b>" i la crida a *bar()* hauria de retornar "<i>bar</i>".

## 6 Haskell

(2 punts)

Recordeu les definicions de les classes *Functor* i *Monad*:

```
class Functor f where
    fmap :: (a → b) → (f a → f b)

class Monad m where
    return :: a → m a
    (≫=) :: m a → (a → m b) → m b
```

Recordeu també les lleis dels functors:

- $fmap \text{id} = \text{id}$
- $fmap(f \cdot g) = fmap f \cdot fmap g$

Considereu aquest tipus de dades per a arbres binaris amb informacions a les fulles:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
deriving (Show)
```

1. Feu que *Tree* sigui instància de *Functor*.
2. Demostreu que la vostra instanciació compleix les lleis dels functors.
3. Feu que *Tree* sigui instància de *Monad* (ignoreu *Applicative*).
4. Escriviu una funció

```
replace :: Eq a ⇒ [(a, b)] → Tree a → Tree (Maybe b)
```

que reemplaci les ocurredies de tipus *a* en un *Tree a* per quelcom de tipus *Maybe b* buscant a la llista associativa. Per exemple,

- $\text{replace } [] (\text{Leaf } 'x') = \text{Leaf } \text{Nothing}$
- $\text{replace } [('x', 3)] (\text{Leaf } 'x') = \text{Leaf } (\text{Just } 3)$
- $\text{replace } [('x', 3)] (\text{Node } (\text{Leaf } 'x') (\text{Leaf } 'y'))$   
 $= \text{Node } (\text{Leaf } (\text{Just } 3)) (\text{Leaf } \text{Nothing})$

Per definir el vostre *replace*, heu d'usar (forçosament) el  $\gg=$  de l'apartat anterior i podeu usar la funció estàndard *lookup* ::  $\text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow \text{Maybe } b$ .

## 7 Compilació

(2 punts)

A continuació teniu un fragment d'una gramàtica ANTLR per la definició de tipus al llenguatge de programació Standard ML:

```
grammar LittleML ;

root      : type EOF
            ;
type      : type ID                                #typeType
           | '(' type (',', type)+ ')' ID      #typeApp
           | type '*' type                      #typePair
           | <assoc=right> type '->' type    #typeArrow
           | '{' record? '}'
           | '(' type ')'
           | TVAR                           #typeParen
           | ID                            #typeName
            ;
record    : ID '::' type (',', record)?
            ;
ID        : [A-Za-z] [A-Za-z0-9]* ;
TVAR      : '\' [A-Za-z0-9]+ ;
WS        : [ \t\n]+ -> skip      ;
```

1. Digueu quins visitadors crea ANTLR per la classe LittleMLVisitor.
2. Digueu quines de les entrades següents són vàlides per a la gramàtica anterior. En cas negatiu, expliqueu perquè; en cas afirmatiu, doneu el seu arbre de parsing.
  - (a) `unsigned_int`
  - (b) `int list tree`
  - (c) `int * float + char`
  - (d) `(int, char) tree`
  - (e) `{x:int, y:float}`
  - (f) `'a 'b -> int`
3. Per a casdavant de les entrades vàlides anteriors, digueu en quin ordre es cridaran els visitadors quan es parseji.
4. Expliqueu raonadament si, tal com es troba, aquesta gramàtica es pot parsejar amb un parser LL(1).

# Llenguatges de Programació, FIB, 17 de gener de 2022

## Possibles solucions

### 1 Pragmàtica dels LPs

Sense els parèntesis hi podria haver ambigüïtats. Per exemple: **if**  $x$   $\text{++ } y$ ; voldria dir **if**  $(x++) \text{ } y$ ; o bé **if**  $(x) \text{ ++ } y$ ?

### 2 $\lambda$ -càlcül (1 punt)

Veremos que  $\text{AND} \equiv \lambda a \ b. a \ b \ (\lambda x \ y. y)$ , evaluando  $(\text{AND} \ u \ v)$  considerando los posibles casos de  $u$  y  $v$ :

$$(\text{AND} \ u \ v) \equiv (\lambda a \ b. a \ b \ (\lambda x \ y. y))u \ v \rightarrow_{\beta} u \ v \ (\lambda x \ y. y)$$

Si  $u$  es *False*, es decir  $u \equiv \lambda x \ y. y$ :

$$u \ v \ (\lambda x \ y. y) \equiv (\lambda x \ y. y) \ v \ (\lambda x \ y. y) \rightarrow_{\beta} \lambda x \ y. y \equiv \text{False}$$

Es decir  $\text{AND False } v = \text{False}$ . Y si  $u$  es *True*, es decir  $u \equiv \lambda x \ y. x$ :

$$u \ v \ (\lambda x \ y. y) \equiv (\lambda x \ y. x) \ v \ (\lambda x \ y. y) \rightarrow_{\beta} v$$

Por tanto, si  $v \equiv \text{False}$ , entonces  $\text{AND True False} = \text{False}$ , y si  $v \equiv \text{True}$ , entonces  $\text{AND True True} = \text{True}$ .

### 3 Subtipus

- En principio, los punteros podrían considerarse covariantes ya que las operaciones de acceso y de modificación del contenido de la dirección apuntada por el puntero preservan la relación de subtipo. Sin embargo, la covarianza de los punteros puede dar problemas de inconsistencia de tipos. Veamos un ejemplo. Supongamos que tenemos las clases *Empleado*, *Vendedor* y *Administrativo*, con *Vendedor*  $\leq$  *Empleado* y *Administrativo*  $\leq$  *Empleado* y definimos la función

```
void P(Empleado* p, Empleado e){  
    *p = e; \  
}
```

El problema es que si los punteros son covariantes entonces, podemos escribir:

```
Vendedor v;  
Administrativo* p1;  
...  
P(p1, v);
```

lo que produciría que  $p1$ , de tipo puntero a *Administrativo* apuntara a un objeto de tipo *Vendedor*.

- Los punteros no se deben de considerar contravariantes. Por ejemplo, si lo fueran, podríamos escribir;

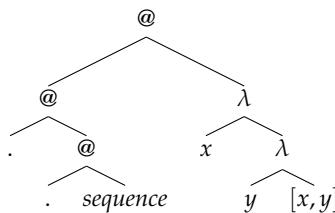
```
Vendedor v;
Administrativo a;
Empleado* p1;
Vendedor* p2;
...
p1 = new Empleado a;
p2 = p1;
v = *p2;
```

lo que produciría que a v se le asignara un objeto de tipo Administrativo.

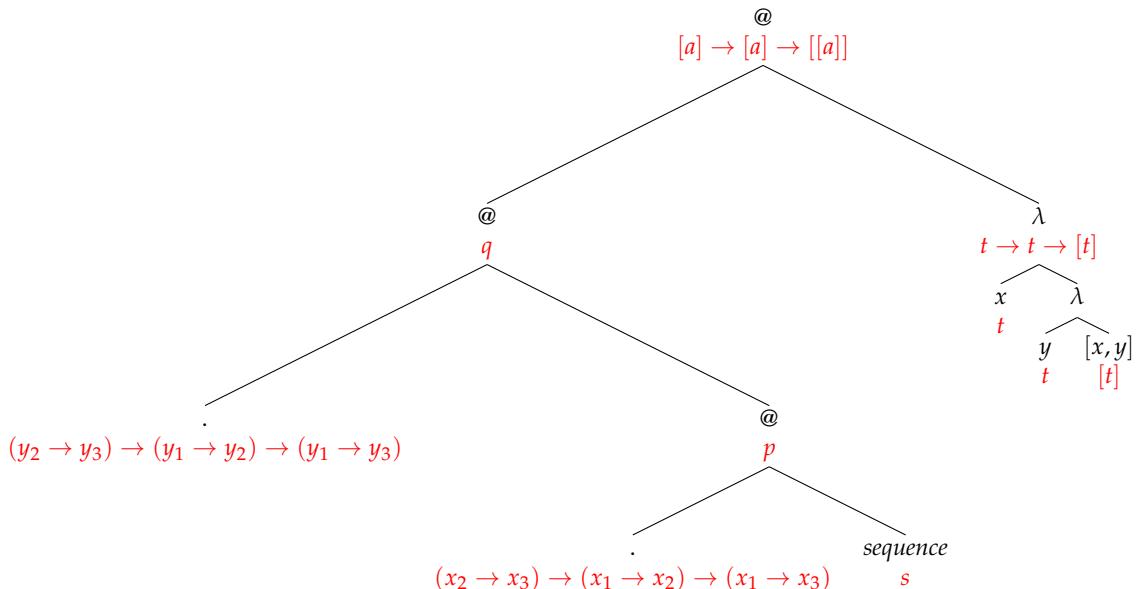
- Desde este punto de vista, lo más razonable sería considerar que los punteros deberían de ser invariantes.

## 4 Inferència de tipus

1. Aquest és l'arbre de sintàxi abstracta de *cartesian*:



2. Aquesta és la seva anotació de tipus. Per no generar tantes restriccions, ja s'ha inferit tribalment el tipus del fill dret de l'arrel.



3. Les restriccions que apareixen són les següents:

$$\begin{aligned}
 (x_2 \rightarrow x_3) \rightarrow (x_1 \rightarrow x_2) \rightarrow (x_1 \rightarrow x_3) &= s \rightarrow p \\
 (y_2 \rightarrow y_3) \rightarrow (y_1 \rightarrow y_2) \rightarrow (y_1 \rightarrow y_3) &= p \rightarrow q \\
 (t \rightarrow t \rightarrow [t]) \rightarrow ([a] \rightarrow [a] \rightarrow [[a]]) &= q
 \end{aligned}$$

4. Resolem les restriccions per trobar el tipus  $s$ :

$$\begin{aligned}
 s &= x_2 \rightarrow x_3 \\
 p &= (x_1 \rightarrow x_2) \rightarrow (x_1 \rightarrow x_3) \\
 p &= y_2 \rightarrow y_3 \\
 y_2 &= x_1 \rightarrow x_2 \\
 y_3 &= x_1 \rightarrow x_3 \\
 q &= (y_1 \rightarrow y_2) \rightarrow (y_1 \rightarrow y_3)) \\
 y_1 \rightarrow y_2 &= t \rightarrow t \rightarrow [t] \\
 y_1 \rightarrow y_3 &= [a] \rightarrow [a] \rightarrow [[a]] \\
 y_1 &= t = [a] \\
 y_2 &= t \rightarrow [t] \\
 y_3 &= [a] \rightarrow [[a]] \\
 x_1 &= t \rightarrow [a] \\
 x_2 &= [t] = [[a]] \\
 x_3 &= [[a]] \\
 s &= [[a]] \rightarrow [[a]].
 \end{aligned}$$

Per tant,  $\text{sequence} :: [[a]] \rightarrow [[a]]$ .

[En realitat,  $\text{sequence} :: (\text{Traversable } t, \text{ Monad } m) \Rightarrow t(m\ a) \rightarrow m(t\ a)$  però això no es podia inferir de l'arbre, calia més context.]

## 5 Python

```

def make_italic(func):    return lambda : "<i>" + func() + "</i>"
def make_bold(func):      return lambda : "<b>" + func() + "</b>"
def make_underline(func): return lambda : "<u>" + func() + "</u>"

```

I refactoritzar mai és dolent:

```

def add_tag(func, tag):
    return "<" + tag + ">" + func() + "</" + tag + ">"

def make_italic(func):    return add_tag(func, "i")
def make_bold(func):      return add_tag(func, "b")
def make_underline(func): return add_tag(func, "u")

```

## 6 Haskell

```

instance Functor Tree where
  fmap f (Leaf x)  = Leaf (f x)
  fmap f (Node e1 e2) = Node (fmap f e1) (fmap f e2)

```

```

instance Monad Tree where
  return = Leaf
  (Leaf x) >>= f = f x
  (Node e1 e2) >>= f = Node (e1 >>= f) (e2 >>= f)

```

```

replace dic e = e >>= f
  where f x = Leaf (lookup x dic)

```

Primera llei dels functors:

- $fmap \text{id} (L x) = L (\text{id} x) = L x = \text{id} (L x) \checkmark$
- $fmap \text{id} (N e1 e2) = N (fmap \text{id} e1) (fmap \text{id} e2) = N e1 e2 = \text{id} (N e1 e2) \checkmark$

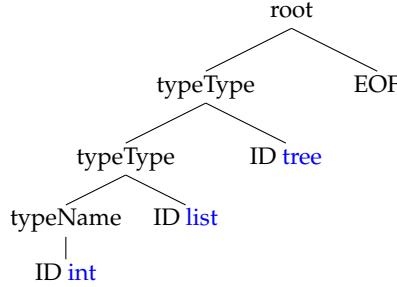
Segona llei dels functors:

- $fmap (f . g) (L x) = L ((f . g) x) = L f (g x) =$   
 $fmap f (L (g x)) = fmap f ((fmap g) (L x)) = ((fmap f).(fmap g)) (L x) \checkmark$
- FALTA FER!!!  $\checkmark$

## 7 Compilació

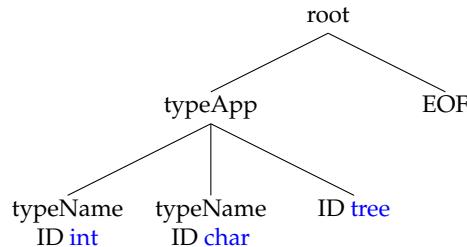
1. visitRoot, visitTypeParen, visitTypeVar, visitTypePair, visitTypeApp, visitTypeRecord, visitTypeName, visitTypeType, visitTypeArrow i visitRecord.
2. (a) no ho és perquè el subtratllat no és a cap token.

(b)

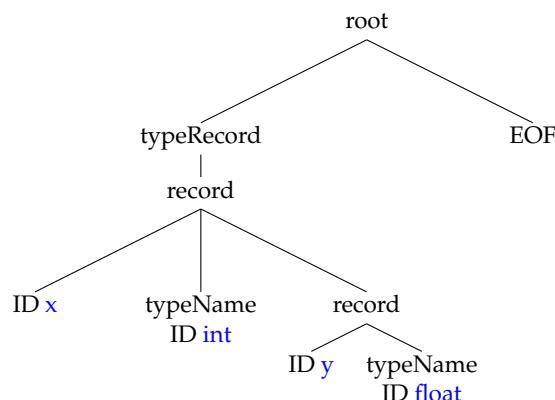


(c) no ho és perquè el símbol de suma no és a cap token.

(d)



(e)



(parèntesis, dos punts i comes no es mostren per concisió)

(f) no ho és perquè la gramàtica impedeix que dos TVARs vagin seguits.

3. Els nodes dels arbres es visiten en preordre:

(b) visitRoot, visitTypeType, visitTypeType, visitTypeName.

(d) visitRoot, visitTypeApp, visitTypeName, visitTypeName.

(e) visitRoot, visitTypeRecord, visitRecord, visitTypeName, visitRecord, visitTypeName.

4. Aquesta gramàtica no es pot parsejar amb un parser LL(1) perquè la regla type té recursivitat per l'esquerra i prefixos comuns.

# Llenguatges de Programació, FIB, 16 de juny de 2021

*L'examen dura tres hores. Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes. Contesteu cada problema en un full diferent i poseu el vostre nom a cada full. Feu bona lletra.*

## 1. Pragmàtica dels LPs (1 punt)

En el seu assaig *Why Pascal is Not My Favorite Programming Language* (1981), Brian Kernighan (autor del famós llibre *The C Programming Language*) critica (entre d'altres) que el llenguatge Pascal no tingui suport per a instruccions tipus **break** o **continue**.

Des del punt de vista de la pragmàtica dels llenguatges de programació, doneu un argument a favor i un argument en contra d'aquesta decisió de disseny.

Limiteu la vostra resposta a 10 línies (menys també pot estar molt bé!).

## 2. Haskell (2 punts)

En matemàtiques, un *monoïde* és una estructura algebraica consistent en un conjunt dotat d'una operació interna associativa i d'un element neutre. En Haskell, la classe *Monoïde* es pot definir així:

```
class Monoïde m where
    neutre :: m
    ( $\diamond$ ) :: m  $\rightarrow$  m  $\rightarrow$  m           -- (la caixa  $\diamond$  és un símbol d'operador binari)
```

Les lleis de la classe *Monoïde* són les següents:

- (1)  $\text{neutre} \diamond x = x$
- (2)  $x \diamond \text{neutre} = x$
- (3)  $x \diamond (y \diamond z) = (x \diamond y) \diamond z$

per a tots  $x, y, z :: m$ .

1. Feu que el tipus **Int** instanciï la classe *Monoïde* de dues maneres diferents.
2. Feu que el tipus llista de  $t$  (`[t]` o **List**  $t$ ) instanciï la classe *Monoïde* i demostreu que la vostra instanciació compleix les lleis dels monoides.
3. Sabent que el tipus  $t$  és un monoïde, feu que el tipus **Maybe**  $t$  instanciï la classe *Monoïde* i demostreu que la vostra instanciació compleix les lleis dels monoides.

## 3. Inferència de tipus (2 punts)

1. Apliqueu l'algorisme d'inferència de tipus per trobar el tipus de la funció  $f$  següent:

$$\begin{aligned} f\ 0\ c &= c\ 1 \\ f\ n\ c &= f\ (n - 1)\ (c\ .\ (*\ n)) \end{aligned}$$

Per a fer-ho, dibuixeu els arbres de sintàxi, etiqueteu els nodes amb els seus tipus i gene-reu i resoleu metodològicament les restriccions de tipus. Expliqueu els passos realitzats.

Suposeu que tots els literals i operadors numèrics són sobre el tipus **Int**.

2. Quin és el tipus de  $f\ n\ \text{id}$ ?
3. I, ja que hi som, què calcula  $f\ n\ \text{id}$ ?

## 4. Lambda càcul

(2 punts)

1. Avalueu aquesta expressió en ordre normal i en ordre aplicatiu:

$$(\lambda x . \lambda y . (\text{ADD } y ((\lambda z . (\text{MUL } x z)) 3))) 7 5$$

Podeu suposar que ADD i MUL són funcions que sumen i multipliquen respectivament.

2. Avalueu aquesta expressió en ordre normal i en ordre aplicatiu:

$$(\lambda y . 5)((\lambda x . x x)(\lambda x . x x))$$

## 5. Compilació

(1 punt)

Escriviu, utilitzant notació ANTLR, una gramàtica LL(1) pel llenguatge dels parèntesis i claudàtors ben tancats.

Per exemple: “[]([()[]])”, “[ ]” i “((()))” pertanyen al llenguatge, però “[ () ]”, “[ () [] ]”, “[ () ]”, “[ ]”, “” i “unicorn” no.

Justifiqueu perquè la vostra gramàtica és LL(1).

## 6. Subtipos y herencia

(2 punts)

Considerad las definiciones siguientes en C++:

<pre>class A { public:     virtual void m1();     A m2(); };  A a; A* p;</pre>	<pre>class B : public A { public:     virtual void m1();     void m3(); };  B b; B* q;</pre>
--	--

Explicad si serían legales las siguientes secuencias de instrucciones:

- (1)  $b.m1(); b.m3();$
- (2)  $b = b.m2(); b.m3();$
- (3)  $a = b; a.m3();$
- (4)  $p = \text{new } B; p \rightarrow m3();$
- (5)  $q = \text{new } B; q \rightarrow m3();$
- (6)  $q = p; q \rightarrow m3();$
- (7)  $p = q; p \rightarrow m3();$

¿Qué método  $m1$  (el de  $A$  o el de  $B$ ) se ejecuta en los casos siguientes? Explicad porqué.

- (8)  $p = \text{new } A; p \rightarrow m1();$
- (9)  $p = \text{new } B; p \rightarrow m1();$
- (10)  $a = b; a.m1();$

# Llenguatges de Programació, FIB, 16 de juny de 2021

## Possibles solucions

### 1. Pragmàtica dels LPs

Pascal fou un LP dissenyat per a l'educació que encoratjava bones pràctiques de programació basades en la programació estructurada. Tenir un sol punt de sortida és un avantatge perquè fa els procediments més entenedors. Tanmateix, algunes situacions habituals són més clares de programar amb **breaks** i **continues** (no cal tant d'aniuament de construccions i s'eviten variables booleanes per controlar el flux d'execució).

[Error comú: Perquè tinc un 8 enllloc d'un 10? Probablement has donat raons a favor i en contra correctes, però has oblidat de posar en context el disseny del llenguatge: Pascal era un LP per a l'ensenyança de la programació estructurada.]

### 2. Haskell

1. Amb la suma:

```
instance Monoide Int where
    neutre = 0
    ( $\diamond$ ) = (+)
```

Amb el producte:

```
instance Monoide Int where
    neutre = 1
    ( $\diamond$ ) = (*)
```

2. Instanciació:

```
instance Monoide [t] where
    neutre = []
    ( $\diamond$ ) = (++)
```

Lleis:

$$\begin{aligned}
(1) \ neutre \diamond x &= [] ++ x = x \checkmark \\
(2) x \diamond neutre &= x ++ [] = x \checkmark \\
(3) x \diamond (y \diamond z) &= x ++ (y ++ z) = (x ++ y) ++ z = (x \diamond y) \diamond z \checkmark
\end{aligned}$$

3. Instanciació:

```
instance Monoid t  $\Rightarrow$  Monoid (Maybe t) where
    neutre = Nothing
    Nothing  $\diamond$  a = a
    a  $\diamond$  Nothing = a
    Just a  $\diamond$  Just b = Just (a  $\diamond$  b)
```

Per brevetat, utilitzarem  $N = \text{Nothing}$  i  $J = \text{Just}$ .

Lleis: La (1) i la (2) són evidents. Per a la (3):

- Si  $x, y$  o  $z$  són  $N$ , es fàcil de veure. Per exemple, quan  $x = N$ :

$$x \diamond (y \diamond z) = N \diamond (y \diamond z) = y \diamond z = (N \diamond y) \diamond z = (x \diamond y) \diamond z \checkmark$$

(Aquí hem usat la propietat de l'element neutre.)

- Si  $x = J a$ ,  $y = J b$  i  $z = J c$ :

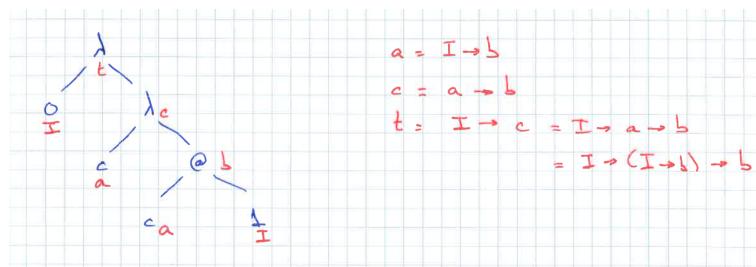
$$x \diamond (y \diamond z) = J a \diamond (J b \diamond J c) = J a \diamond J (b \diamond c) = J (a \diamond (b \diamond c))$$

$$= J ((a \diamond b) \diamond c) = J (a \diamond b) \diamond J c = (J a \diamond J b) \diamond J c = (x \diamond y) \diamond z \checkmark$$

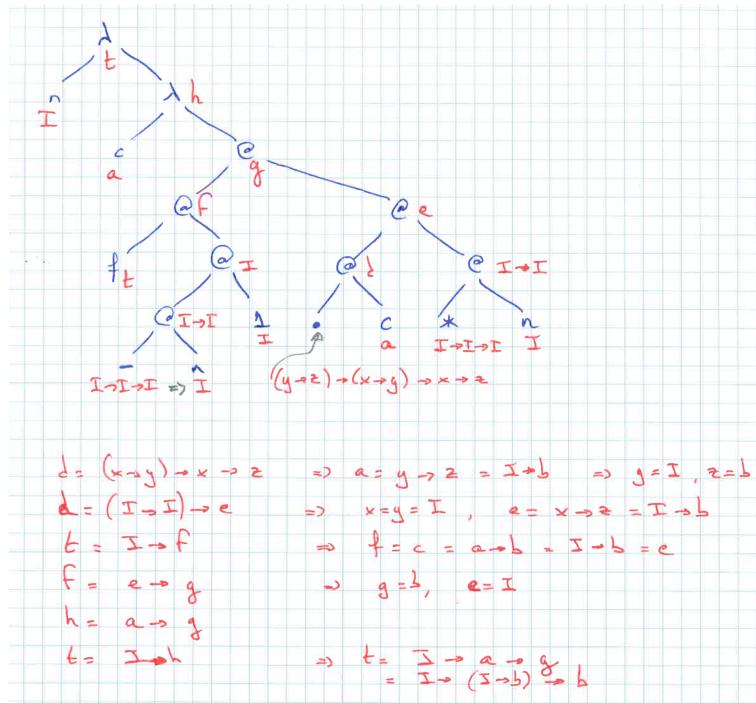
(Aquí hem usat el fet que  $t$  és un monoïde.)

### 3. Inferència de tipus

1. Comencem amb l'arbre de la primera definició i apliquem l'algorisme:



Ens surt que  $f :: t$  amb  $t = I \rightarrow (I \rightarrow b) \rightarrow b$ . Per tal de veure si podem trobar què és  $b$ , continuem amb l'arbre de la segona definició:



Vaja, no hem guanyat gaire...  $b$  és encara una variable de tipus.

2. Amb  $f :: I \rightarrow (I \rightarrow b) \rightarrow b$  i **id** ::  $s \rightarrow s$ , ara podem inferir que  $f \circ \text{id} :: I$ .

3.  $f \circ \text{id}$  calcula el factorial d'un enter  $n$  (per a valors negatius es penja).

## 4. Lambda càcul

$$1. (\lambda x . \lambda y . (\text{ADD } y ((\lambda z . (\text{MUL } x z)) 3))) 7 5$$

- Ordre normal:

$$\begin{aligned} & (\lambda x . \lambda y . (\text{ADD } y ((\lambda z . (\text{MUL } x z)) 3))) 7 5 \\ & \rightarrow_{\beta} \lambda y . (\text{ADD } y ((\lambda z . (\text{MUL } 7 z)) 3)) 5 \\ & \rightarrow_{\beta} \text{ADD } 5 ((\lambda z . (\text{MUL } 7 z)) 3) \\ & \rightarrow_{\beta} \text{ADD } 5 (\text{MUL } 7 3) \\ & \rightarrow_{\beta} \text{ADD } 5 21 \rightarrow_{\beta} 26. \end{aligned}$$

- Ordre aplicatiu:

$$\begin{aligned} & (\lambda x . \lambda y . (\text{ADD } y ((\lambda z . (\text{MUL } x z)) 3))) 7 5 \\ & \rightarrow_{\beta} (\lambda x . \lambda y . (\text{ADD } y (\text{MUL } x 3))) 7 5 \\ & \rightarrow_{\beta} \lambda y . (\text{ADD } y (\text{MUL } 7 3)) 5 \\ & \rightarrow_{\beta} \lambda y . (\text{ADD } y 21) 5 \\ & \rightarrow_{\beta} \text{ADD } 5 21 \\ & \rightarrow_{\beta} 26. \end{aligned}$$

$$2. (\lambda y . 5)((\lambda x . x x)(\lambda x . x x))$$

- Ordre normal:

$$(\lambda y . 5)((\lambda x . x x)(\lambda x . x x)) \rightarrow_{\beta} 5.$$

- Ordre aplicatiu:

$$\begin{aligned} & (\lambda y . 5)((\lambda x . x x)(\lambda x . x x)) \\ & \rightarrow_{\beta} (\lambda y . 5)((\lambda x . x x)(\lambda x . x x)) \\ & \rightarrow_{\beta} (\lambda y . 5)((\lambda x . x x)(\lambda x . x x)) \\ & \rightarrow_{\beta} \dots \end{aligned}$$

## 5. Compilació

```
grammar BenPosats;

root : par root
      |
      ;

par  : '(' root ')'
      | '[' root ']'
      ;
```

La gramàtica és LL(1) perquè a `root` es pot triar entre les dues produccions en funció de si el caràcter següent és un parèntesi o claudàtor obert (primera) o no (segona) i perquè a `par` es pot triar entre les dues produccions en funció de si el caràcter següent és un parèntesi obert (primera) o un claudàtor obert (segona).

## 6. Subtipos y herencia

- (1) Sí, serien legals.
- (2) No, perquè `m2` retorna un `A` i no es pot assignar un `A` a un `B`.
- (3) No, tot i que l'assignació sí és legal. La comprovació de tipus en compilari farà que la invocació a `m3` doni un error.

- (4) No, per les mateixes raons que l'apartat 3.
- (5) Sí, seria legal.
- (6) No, no es pot assignar un punter a un  $B$  a un  $A^*$ .
- (7) No, per les mateixes raons que l'apartat 3 i 4.
- (8) L' $m1$  d' $A$ .
- (9) L' $m1$  de  $B$  (compila bé perquè  $A$  té  $m1$ , però l'objecte apuntat és instància de  $B$ ).
- (10) Al convertir el  $b$  en  $a$ , queda un  $A$ , és a dir que es perd tota la informació de  $B$ . Per tant, es crida al  $m1$  de  $A$ .

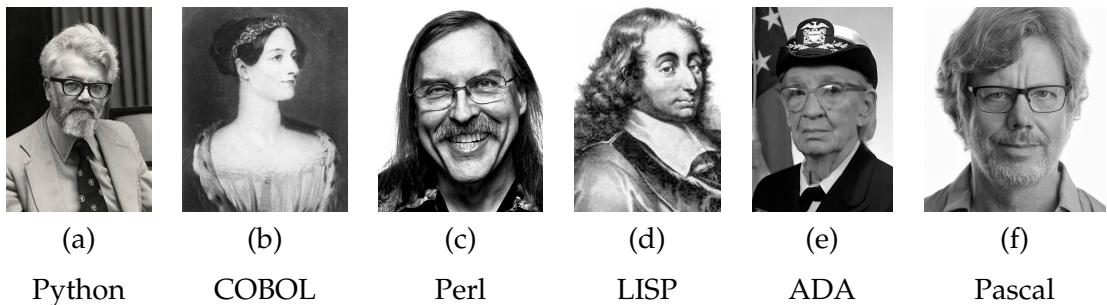
# Llenguatges de Programació, FIB, 18 de gener de 2021

*L'examen dura dues hores i mitja. Es valorarà la concisió, claredat i brevetat a més de la completenessa 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 puntuuen igual.*

## 1. El Trivial d'LP

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

1. Aparellau cares i llenguatges de programació:



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 *currificar* 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) → (fa → fb)
```

## 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] ;
```

continuem 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*.

# Llenguatges de Programació, FIB, 18 de juny de 2020

## Possibles solucions

### 1. El Trivial d'LP

1. a: LISP, b: ADA, c: Perl, d: Pascal, e: COBOL, f: Python.
2. FORTRAN.
3. COBOL.
4. LISP o Prolog. [Avui en dia Python.]
5. Un llenguatge de programació és *type safe* quan les úniques operacions que es poden aplicar a les dades són les que permeten el tipus de les dades. En aquest sentit, Java està dissenyat per ser *type safe*. [Realment és molt més complicat, però caldria molt d'espai i tecnicismes per explicar-ho.]
6. 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).
7. C [amb Assembler].
8. Simula 67 [tot i que hi ha trets de l'OOP moderna que no hi eren].
9. Una funció que transforma una funció en una altra funció amb determinada funcionalitat afegida, modificant el seu comportament original. / Una notació per tornar noves funcions a partir de funcions.
10. Smalltalk.
11. ADA.
12. Una funció que capture el seu context lèxic en el moment de la seva creació.
13. Una classe abstracta és una classe que té alguns mètodes declarats però no implementats.
14. Java utilitzà pas per valor (en el cas d'objectes, passa per valor referències als objectes).
15. La Màquina de Turing és un model matemàtic. Com a tal, no té un cost.

## 2. Haskell

TreeFold:

```
treeFold _ z Empty = z
treeFold f z (Node x l r) = f x (treeFold f z l) (treeFold f z r)
```

Funcions:

```
size = treeFold plusOne 0
      where plusOne _ x y = x + y + 1
```

```
height = treeFold maxPlusOne 0
      where maxPlusOne _ x y = 1 + max x y
```

```
treeMap f = treeFold build Empty
      where build x l r = Node (f x) l r
```

```
inOrder = treeFold cat []
      where cat x l r = l ++ x : r
```

```
isBST = isSorted . inOrder
      where isSorted xs = and (zipWith(<) xs) (tail xs)
```

Instanciació:

```
instance Functor Tree where
```

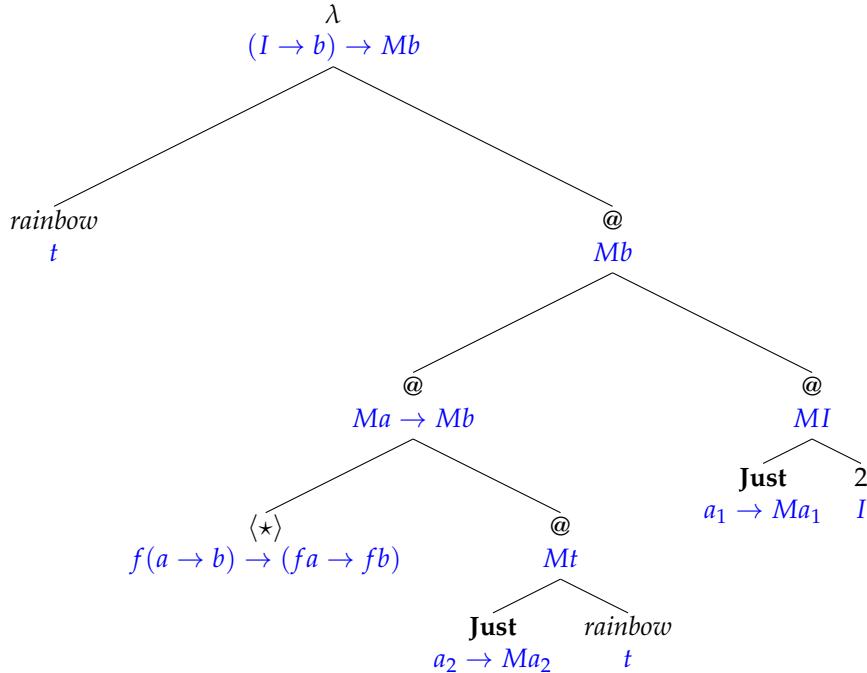
```
  fmap = treeMap
```

```
instance Show a => Show (Tree a) where
```

```
  show = treeFold sh "*"
      where sh x l r = concat ["<", show x, " ", l, " ", r, ">"]
```

### 3. Inferència de tipus

Un cop dibuixat l'arbre de sintàxi abstracta (negre), etiquetem les fulles amb els seus tipus (blau) i, de les fulles cap a l'arrel podem inferir tots els tipus dels nodes.



Hem usat  $I$  i  $M$  com a abreviatura d'**Int** i de **Maybe** i hem trobat que  $a_1 = I = a$ ,  $a_2 = t = a \rightarrow b$  i  $f = M$ . Per tant, tenim  $\text{unicorn} :: (\text{Int} \rightarrow b) \rightarrow \text{Maybe } b$ .

### 4. Python

- Donat un natural  $n$ ,  $\text{mystery1}(n)$  retorna el factorial de  $n$ .
- Donat un natural  $n$ ,  $\text{mystery2}(n)$  retorna l' $n$ -èsim nombre triangular, és a dir,  $\sum_{i=1}^n i = n(n+1)/2$ .
- Donat un natural  $n$ ,  $\text{mystery3}(n)$  retorna  $n + 1$ .
- Donat un natural  $n$ ,  $\text{mystery4}(n)$  retorna la llista  $[n, n - 1, n - 2, \dots, 1]$ .
- Donat un natural  $n$ ,  $\text{mystery5}(n)$  retorna una aproximació de  $\pi$  (més exacta quan més gran sigui la  $n$ ).
- Donat un natural  $n$ ,  $\text{mystery6}(n)$  retorna les  $2^n$  llistes de  $n$  zeros o uns.

El cost de  $\text{mystery4}(n)$  és  $O(n^2)$  perquè cada concatenació té un cost lineal.

**Error freqüent:** No especificar que  $n$  ha de ser un natural. Amb un nombre negatiu o real, les funcions es penjarien. Amb un paràmetre d'un altre tipus, petarien.

## 5. Compilació

1. Una gramàtica és ambigua si una mateixa expressió pot tenir dues derivacions diferents.

2. Hi ha moltes solucions possibles, però una que modifica mínimament la segona solució de l'enunciat i és fàcilment entendible seria:

**grammar Expr;**

*prog* : *expr EOF* ;

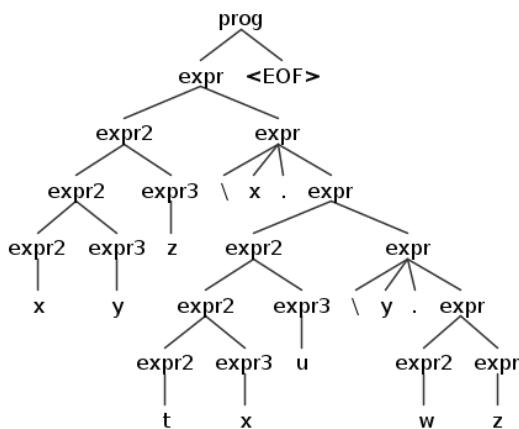
*expr* : *VAR*  
   | '<\>' *VAR* '!' *expr*  
   | '(' *expr* ')'  
   | *expr2 expr*  
   ; ;

*expr2* : *VAR*  
   | '(' *expr* ')'  
   | *expr2 expr3*  
   ; ;

*expr3* : *VAR*  
   | '(' *expr* ')'  
   ; ;

*VAR* : [a-z]  
   ; ;

Que per l'entrada  $x \ y \ z \ \backslash x . t \ x \ u \ \backslash y . w \ z$  generaria:



La idea és que *expr* pugui derivar en qualsevol expressió, *expr2* pot derivar en una seqüència d'aplicacions que no conté cap lambda-expressió a nivell extern, i *expr3* és encara més restringit i no conté ni lambda-abstraccions ni aplicacions a nivell extern, o sigui és simplement un identificador o una expressió encapsulada entre parèntesis.

# Llenguatges de Programació, FIB, 18 de juny de 2020

Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes.  
Tots els problemes puntuuen igual.

## 1. $\lambda$ -càcul

1. Considereu les funcions següents:

$$S = \lambda f g x. f x (g x)$$

$$K = \lambda x y. x$$

$$I = \lambda x. x$$

Demostreu que  $S K K = I$ .

2. Recordeu que, al  $\lambda$ -càcul, els valors fals i cert es defineixen amb  $F = \lambda x y. y$  i  $T = \lambda x y. x$  respectivament.

Trobeu una definició ben senzilla per la funció  $C$  (abreviatura de condicional) que, aplicada a tres valors, retorna el segon si el primer és cert i retorna el tercer si el primer és fals (és a dir, un *if-then-else*).

Mostreu que la vostra definició és correcta.

## 2. Haskell

Digueu, de forma raonada i utilitzant reescritura, quan val *unicorn*:

```
unicorn = do
    x ← [1, 2, 3]
    y ← [4, 5]
    return x
```

Compte! La resposta correcta no és  $unicorn = [1, 2, 3]$ . Recordeu que les llistes són mònades. Concretament:

```
instance Monad [] where
    return x = [x]
    xs ≫= f = concat (map f xs)           -- = concatMap f xs
```

## 3. Inferència de tipus

Inferiu el tipus més general de la funció *rainbow* sabent que  $frozen :: \text{Int} \rightarrow \text{Int}$ .

```
rainbow x = do
    y ← x
    return (frozen y)
```

Per a fer-ho, dibuixe el seu arbre de sintàxi, etiqueteu els nodes amb els seus tipus i genereu les restriccions de tipus i classes. Resoleu-les per obtenir la solució i assenyaleu el resultat final amb un quadre.

## 4. Compilació

Es vol extender la pràctica d'aquest curs (SkylineBot) per incorporar dos nous operadors:

- $s_1 - s_2$

Aquest operador substreu l'skyline  $s_2$  a l'skyline  $s_1$ , i té la mateixa prioritat que l'operador de suma.

- $s[x_1:x_2]$

Aquest operador selecciona el segment de l'skyline  $s$  que està entre les posicions  $x_1$  i  $x_2$ . També es permet aplicar l'operador quan una de les dues posicions sigui absent (però no les dues), amb la semàntica habitual d'indexació de vectors:  $s[:x_2]$  selecciona l'skyline  $s$  desde l'inici fins a la posició  $x_2$ , i  $s[x_1:]$  selecciona l'skyline  $s$  desde el final fins a la posició  $x_1$ .

Feu una extensió al llenguatge d'operacions sobre skylines per poder parsejar els dos operadors anteriors. Heu d'incloure tot el vostre fitxer de gramàtica (.g) ressaltant-hi clarament (per exemple, amb colors o comentaris) els canvis efectuats.

## 5. Subtipos en C++

Como ya debéis saber, un constructor de tipos  $C$  es covariante si cuando  $T_1$  es subtipo de  $T_2$ , se tiene que  $C\langle T_1 \rangle$  es subtipo de  $C\langle T_2 \rangle$ .  $C$  es contravariante si cuando  $T_1$  es subtipo de  $T_2$ , se tiene que  $C\langle T_2 \rangle$  es subtipo de  $C\langle T_1 \rangle$ . Y  $C$  es invariante si no es ni covariante ni contravariante.

1. ¿Podemos considerar covariante el constructor  $\text{vector}\langle T \rangle$  ?

En caso que creáis que sí, argumentad por qué lo creéis. Si creéis que no, poned un ejemplo de una función  $f$  con (al menos) un parámetro de tipo  $\text{vector}\langle T_2 \rangle$  que nos pudiera dar problemas si **vector** fuera covariante.

2. ¿Podemos considerar contravariante el constructor  $\text{vector}\langle T \rangle$  ?

En caso que creáis que sí, argumentad por qué lo creéis. Si creéis que no, poned un ejemplo de una función  $f$  con (al menos) un parámetro de tipo  $\text{vector}\langle T_1 \rangle$  que nos pudiera dar problemas si **vector** fuera contravariante.

## 6. Python

Si executem el següent programet, el resultat és [1, 2, 3].

```
def func1():
    def f1(i):
        t = i
        return lambda: t

    t = 0
    return [f1(i) for i in [1, 2, 3]]

fs = func1()
print([f() for f in fs])
```

En canvi, si executem aquest altre programet, el resultat és [3, 3, 3].

```
def func2():
    def f2(i):
        nonlocal t
        t = i
        return lambda: t

    t = 0
    return [f2(i) for i in [1, 2, 3]]

fs = func2()
print([f() for f in fs])
```

Expliqueu per què es dóna aquesta diferència.

# Llenguatges de Programació, FIB, 18 de juny de 2020

## Possible solució

### 1. $\lambda$ -càlcul

1. Tinem:

$$\begin{aligned} S \ K \ K &= ((\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)) \\ &= ((\lambda yz.(\lambda xy.x)z(yz))(\lambda xy.x)) \\ &= \lambda z.(\lambda xy.x)z((\lambda xy.x)z) \\ &= \lambda z.(\lambda y.z)((\lambda xy.x)z) \\ &= \lambda z.z \\ &= I \end{aligned}$$

2. Agafem  $C = I$  (la identitat!).

Comprovem-ho pels dos casos de cert i fals:

- $C \top a b = I (\lambda xy.x) a b = (\lambda xy.x) a b = a$ , tal com cal.
- $C \bot a b = I (\lambda xy.y) a b = (\lambda xy.y) a b = b$ , tal com cal.

### 2. Haskell

```
unicorn = do
    x ← [1, 2, 3]
    y ← [4, 5]
    return x
= [1, 2, 3] >>= (λx → ([4, 5] >>= (λy → return x)))
= [1, 2, 3] >>= (λx → concat (map (λ_ → [x]) [4, 5]))
= [1, 2, 3] >>= (λx → concat [[x], [x]]))
= [1, 2, 3] >>= (λx → [x, x]))
= concat (map (λx → [x, x]) [1, 2, 3])
= concat [[1, 1], [2, 2], [3, 3]]
= [1, 1, 2, 2, 3, 3]
```

[Nota: La llista de comprensió

```
unicorn = [x | x ← [1, 2, 3], y ← [4, 5]]
```

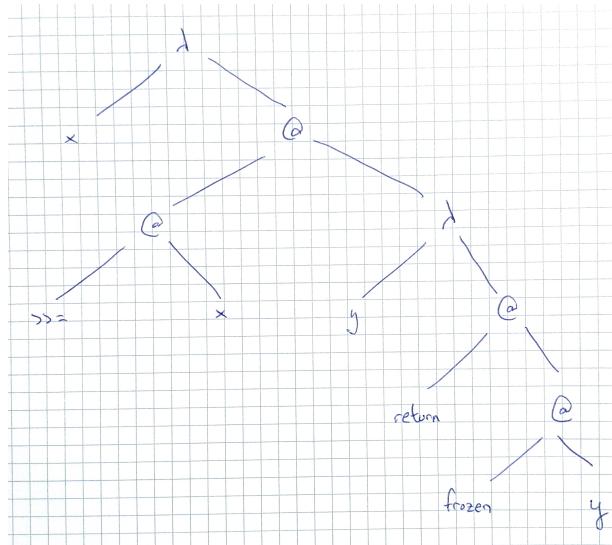
també és equivalent al programa anterior, perquè les llistes per comprensió no són altre cosa que sucre sintàctic al voltant de les llistes com a mònades.]

### 3. Inferència de tipus

Primer, convertim el programa amb notació **do** a notació funcional:

$$\text{rainbow } x = x \gg= \lambda y \rightarrow \text{return} (\text{frozen } y)$$

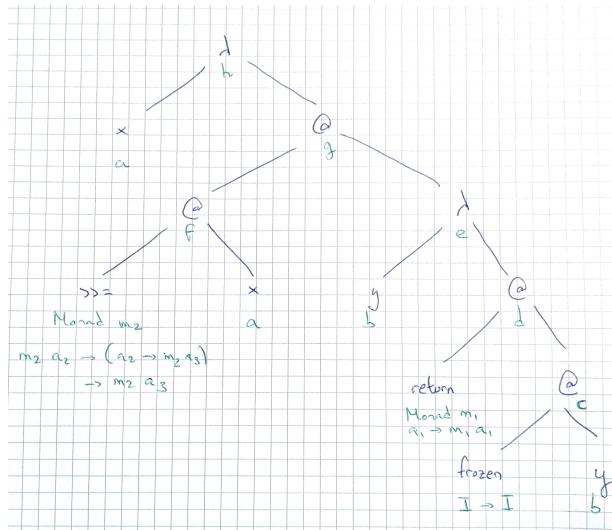
Ara, podem escriure el seu arbre de sintàxi abstracta:



Tenint en compte que ja sabem aquests tipus

$$\begin{aligned} \text{frozen} &:: \text{Int} \rightarrow \text{Int} \\ \text{return} &:: \text{Monad } m \Rightarrow a \rightarrow m a \\ (\gg=) &:: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b \end{aligned}$$

ara podem etiquetar els nodes de l'arbre amb els seus tipus:



La resolució des de les fulles cap a l'arrel permet obtenir el tipus de  $h$  i concloure que

$$\text{unicorn} :: \text{Monad } m \Rightarrow m \text{ Int} \rightarrow m \text{ Int}$$

## 4. Compilació

```
grammar SkyLine;

root : (assgn | expr) EOF ;

assgn : ID ':=' expr ;

expr : '(' expr ')'
    | '-' expr           // mirror
    | expr '*' expr     // intersection
    | expr '*' NUM      // repeat
    | expr ('+'|'-') expr // union -- substraction
    | expr '-' NUM      // shift
    | expr '[' (NUM ':' NUM? | ':' NUM ) ']' // selection
    | buildings
    | rbuildings
    | building
    | ID
    ;

buildings : '[' building (',', building)* ']' ;
rbuildings : '{', NUM ',', NUM ',', NUM ',', NUM '}' ;
building : '(' NUM ',', NUM ',', NUM ')' ;

NUM : [0-9]+ ;
ID : [a-zA-Z] [a-zA-Z0-9]* ;
WS : [ \n\t]+ -> skip ;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
```

## 5. Subtipos en C++

1. **vector** no puede ser covariante. Por ejemplo, si tenemos que las clases *vendedor* y *contable* son subclase de *empleado* y tenemos la operación:

```
void asignar (vector<empleado>& v, empleado e, int n) {
    v[n] = e;
}
```

Si **vector** fuera covariante, *v1* es de tipo **vector**<*vendedor*> y que *c2* es de tipo *contable* y hacemos la llamada:

```
asignar(v1, c, 0);
```

lo que estaría permitido la que ya que **vector**<*vendedor*> sería subtipo de **vector**<*empleado*> y *contable* es subtipo de *empleado*. Sin embargo, la ejecución de esta llamada produciría la desagradable consecuencia de que un vector de vendedores contendría un contable en la posición 0.

2. **vector** no puede ser contravariante. Por ejemplo, si tenemos la operación:

```
vendedor valor (vector<vendedor> v, int n) {
    return v[n];
}
```

y hacemos una llamada a *valor*(*v2*,0), siendo *v2* de tipo **vector**<*empleado*>, lo que estaría permitido al ser vector contravariante, nos devolvería el *v2*[0], que en general será un *empleado*, pero que podría ser un *contable*.

## 6. Python

Primer de tot fixem-nos que en tots dos casos, les crides a  $func1$  i a  $func2$  retornen una llista amb tres funcions, resultat d'aplicar  $f1$  als valors 1, 2 i 3.

- Primer cas: Resultat [1, 2, 3]

En aquest cas les funcions que formen part de la llista retornada per  $func1$  són funcions que retornen el valor de la *variable local* a  $f1$   $t$ , prèvia assignació del valor d' $i$  a  $t$ . Com aquesta variable és local a  $f1$ , a cada crida es *crea de nou*, sent completament independent de les variables locals creades en altres crides a  $f1$ .

- Segon cas: Resultat [3, 3, 3]

En aquest cas les funcions que formen part de la llista retornada per  $func2$  són funcions que retornen el valor de la *variable capturada* (no local)  $t$ , prèvia assignació del valor d' $i$  a  $t$ . Com aquesta variable és no local a  $f2$ , i com les funcions en Python són clausures,  $t$  és una variable *compartida* en cada crida a  $f2$ . Fixeu-vos que el valor retornat per cada crida a  $f2$ ,  $lambda: t$ , fa referència a la *mateixa* variable  $t$ . Així doncs, la darrera crida assigna el valor 3 a  $t$  i aquest és el valor final que tindrà per a cada execució de les funcions retornades per  $f2$ .

# Llenguatges de Programació, FIB, 14 de gener de 2020

Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes. No es pot consultar cap material addicional. Feu bona lletra.

Cada problema compta dos punts.

Solucioneu cada problema en un full different.

Temps: tres hores.

## 1 Inferència de tipus

Un programador novell en Haskell ha escrit aquesta funció *reves* per retornar una llista del revés:

```
reves []      = []
reves (x:xs) = reves xs
```

- a) Inferiu el tipus més general de la funció *reves*. Per a fer-ho, dibuixeu l'arbre de sintàxi de les expressions, etiqueteu els nodes amb els seus tipus i genereu les restriccions de tipus. Resoleu-les per obtenir la solució i assenyaleu el resultat final amb un quadre.
- b) A la vista del tipus obtingut, quina conclusió hauria de treure'n el programador novell?
- c) Si el programador hagués especificat que *reves* ::  $[a] \rightarrow [a]$ , la comprovació de tipus hauria detectat algun error?

## 2 Haskell

Recordeu que la classe **Functor** ve definida per

```
class Functor f where
    fmap :: (a → b) → (f a → f b)
```

i que les lleis dels functors són:

1.  $fmap id \equiv id$
2.  $fmap(g1 . g2) \equiv fmap g1 . fmap g2$

Recordeu també que la classe **Monad** ve definida per

```
class Monad m where
    return :: a → m a
    (>>) :: m a → (a → m b) → m b
```

i que les lleis de les mònades són:

1.  $return x \gg= f \equiv f x$
2.  $m \gg= return \equiv m$
3.  $(m \gg= f) \gg= g \equiv m \gg= (\lambda x \rightarrow f x \gg= g)$

(Ignoreu l'operador  $\gg$  i que les mònades també ha de ser aplicatius i functors.)

- a) Feu que el tipus polimòrfic **Maybe** sigui instància no tribal de la classe **Functor** i demostreu que la vostra instanciació compleix les lleis dels functors.
- b) Feu que el tipus polimòrfic **Maybe** sigui instància no tribal de la classe **Monad** i demostreu que la vostra instanciació compleix les lleis de les mònades.

### 3 Python

a) Què escriu aquest programa en Python?

```
fila = [0, 0, 0, 0]
matriu = [ fila , fila , fila , fila ]
print(matriu [0][0], matriu [0][1], matriu [1][0])
matriu [0][0] = 2
print(matriu [0][0], matriu [0][1], matriu [1][0])
```

b) Què escriu aquest altre programa en Python?

```
def foo(k): k = ["banana"]

def bar(c): c.append("rainbow")

q = ["unicorn"]
foo(q)
bar(q)
print(q)
```

c) Escriviu un generador *nombres(i)* que generi la seqüència infinita  $i, i + 1, i + 2, \dots$

d) Escriviu un generador *filtre (s, p)* que, donats una seqüència *s* i un predicat *p*, generi la seqüència dels elements de *s* que compleixen el predicat *p*.

e) Resumiu, en una línia, les característiques més importants del sistema de tipus de Python.

## 4 Herència en Java i C++

Considereu aquest esquelet de programa en Java:

```
class C {
    int x;
    void m(int y) { ... }
}

class C1 extends C {
    int x1;
}

class C2 extends C1 {
    int x2;
}

class C3 extends C {
    void m(int y) { ... }
}
```

Suposeu que totes les classes, mètodes i atributs són visibles.

- a) Suposant que s'han declarat les variables  $a$ ,  $a1$ ,  $a2$  i  $a3$  com variables de tipus  $C$ ,  $C1$ ,  $C2$  i  $C3$  respectivament, digueu si alguna de les assignacions següents seria incorrecta:

```
a1 = a2;
a2 = a;
a3 = a;
a = a3;
```

- b) Suposem ara que *després* de l'anterior seqüència d'assignacions (una vegada eliminades les assignacions incorrectes) tenim les instruccions:

```
a1.x1 = 3;
a1.x2 = 5;
a2.x2 = 4;
a2.x = 7;
a2.m(7);
a3.m(7);
a.m(7);
```

Alguna d'aquestes instruccions seria incorrecta?

- c) Si les dues darreres instruccions són correctes, quin  $m$  es cridaria en cada cas? El de la classe  $C$  o el de  $C3$ ?

[Continua a la pàgina següent.]

Considereu ara aquest esquelet de programa en C++:

```
class C {
    int x;
    void m(int y);
}

class C1: C {
    int x1;
}

class C2: C1 {
    int x2;
}

class C3: C {
    void m(int y);
}
```

Suposeu que totes les classes, mètodes i atributs són visibles.

d) Suposant que s'han declarat les variables  $a$ ,  $a1$ ,  $a2$  i  $a3$  com variables de tipus  $C$ ,  $C1$ ,  $C2$  i  $C3$  respectivament, digueu si alguna de les assignacions següents seria incorrecta:

```
a1 = a2;
a2 = a;
a3 = a;
a = a3;
```

e) Suposem ara que *després* de l'anterior seqüència d'assignacions (una vegada eliminades les assignacions incorrectes) tenim les instruccions:

```
a1.x1 = 3;
a1.x2 = 5;
a2.x2 = 4;
a2.x = 7;
a2.m(7);
a3.m(7);
a.m(7);
```

Alguna d'aquestes instruccions seria incorrecta?

f) Si les dues darreres instruccions són correctes, quin  $m$  es cridaria en cada cas? El de la classe  $C$  o el de  $C3$ ?

## 5 Compilació

Dóneu (en format .g) l'especificació lèxica i sintàctica d'un compilador per a un llenguatge de gestió de fitxers i directoris. El llenguatge ha de permetre crear fitxers i directoris, copiar fitxers de diverses formes, moure fitxers i directoris, i preguntar per profunditat i amplada respecte el nombre de línies.

Per exemple, aquest és un exemple d'entrada:

```
F1 := mkfile("fitxer")
    // F1 és el fitxer amb nom "fitxer".
    // s'assumeix que l'usuari li posaria contingut.

F2 := F1 ; F1
    // F2 és un nou fitxer que conté dues còpies concatenades
    // -una darrera l'altra- del fitxer F1

dir := mkdir("directori")
    // dir és el directori amb nom "directori".

move(move(dir,F1),F2)
    // moure el fitxer F1 (primer) i el F2 (després) al director dir

F3 := F2 ^ 3
    // F3 es un nou fitxer que és una concatenació del fitxer F2 tres vegades

F4 := F1 <-> F2
    // F4 és un nou fitxer que surt de fusionar línies
    // (primera amb primera, segona amb segona, etc, ...) de F1 i F2

F5 := (F1;F1) <-> (F2;F1)^3
    // exemple d'expressió complexa

length(F1)
    // mostra el nombre de línies de F1

width(F1)
    // mostra la màxima amplada -linia amb nombre màxim de paraules- de F1

maxlength(dir)
    // mostra la màxima length respecte tots els fitxers que pengen de dir

maxwidth(dir)
    // mostra la màxima amplada respecte tots els fitxers que pengen de dir
```



# Llenguatges de Programació, FIB, 14 de gener de 2020

Es valorarà l'eficiència, la concisió, la claredat i la brevetat a més de la completenessa i l'exactitud de les respostes. No es pot consultar cap material addicional. Feu bona lletra.

Aquesta part és pels estudiants que han demanat avaluació única.

Temps: una hora.

## SimpleLP

SimpleLP és un llenguatge de programació imperativa molt simple. Per exemple, aquest és un programa en SimpleLP que correspon a l'algorisme d'Euclides per calcular el màxim comú divisor de 105 i 252 (que és 21):

```
a := 105
b := 252
while a ≠ b do
    if a < b then b := b - a
    else a := a - b
end
```

En SimpleLP, l'únic tipus de dades existent són els enters, amb operacions aritmètiques de suma i resta. També hi ha operacions relacionals per a different-de i menor-que; aquestes retornen 0 per a fals i 1 per a cert. El llenguatge admet variables, amb un valor de zero per defecte. Les instruccions són l'assignació, la composició seqüencial de diverses instruccions, el condicional if-then-else i la iteració while.

L'arbre de sintàxi abstracta de SimpleLP està descrit amb aquestes estructures en Haskell:

```
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
  -- iteratiu
```

Per exemple, aquest és l'arbre de sintàxi abstracta que correspon al programa anterior:

```
euclides = 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"))))
        )
    )
]
```

Una taula de símbols descriu l'estat de la memòria associant els identificadors de les variables amb els seus valors. La taula de símbols és de tipus *Mem* i ofereix aquestes operacions (ja implementades amb arbres AVL):

```
-- retorna una taula de símbols buida
empty :: Mem

-- insereix (o canvia si ja hi era) un identificador amb el seu valor
update :: Mem → String → Int → Mem

-- consulta el valor d'una clau en una taula de símbols
search :: Mem → String → Maybe Int

-- retorna (en ordre lexicogràfic) la llista de claus en una taula de símbols
keys :: Mem → [String]
```

La vostra feina:

- a) Escriu una funció *eval* :: *Expr* → *Mem* → **Int** que avalui una expressió en un estat de la memòria.
- b) Escriu una funció *exec* :: *Instr* → *Mem* → *Mem* que retorni l'estat final de la memòria després d'executar una instrucció partint d'un estat inicial de memòria donat. Per exemple, *exec euclides empty* hauria de retornar una taula de símbols on "a" i "b" valen 21 ambdós.
- c) Digueu en quin cas es podria donar que la vostra funció *exec* no acabés.
- d) Escriu una acció *run* :: *Instr* → **IO** () que escrigui (en ordre lexicogràfic) el valor final de cada variable després d'executar una instrucció partint d'una memòria buida. Per exemple, *run euclides* hauria d'escriure

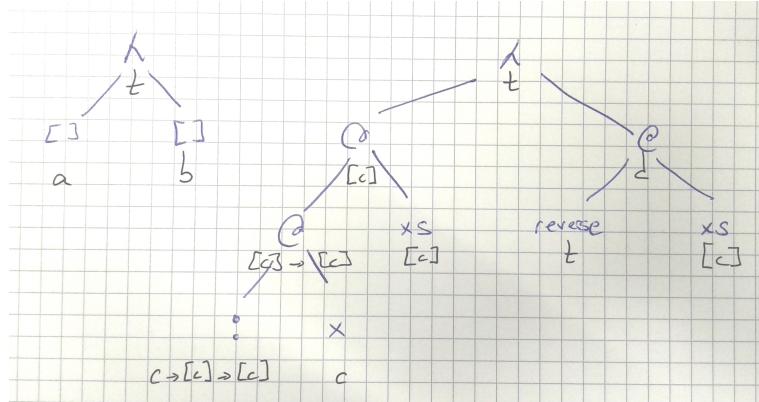
```
a -> 21
b -> 21
```

# Llenguatges de Programació, FIB, 14 de gener de 2020

## Possibles solucions

### 1. Inferència de tipus (1'5 punts)

a) Aquest és l'arbre etiquetat amb els tipus, aplicant les unificacions trivials de baix a dalt:



Les equacions que s'obtenen són:

$$t = [a] \rightarrow [b]$$

$$t = [c] \rightarrow d$$

Per tant,  $a = c$  i  $d = [b]$  i llavors  $t = [a] \rightarrow [b]$  no es pot simplificar més.

b) El programador hauria d'adonar-se que ha comès un error perquè revessar una llista hauria de ser de tipus  $[a] \rightarrow [a]$ .

c) Com que el tipus obtingut sense la declaració de tipus és més general que l'inferit, la comprovació de tipus no hagués detectat cap error, senzillament hauria identificat que  $a = b$ .

### 2. Lleis de functors i mònades en Haskell

a)

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

1. La primera llei es pot comprovar amb dos casos: valors **Nothing** i valors **Just x**:

- $fmap id \text{ Nothing} \equiv \text{ Nothing} \equiv \text{id Nothing} \checkmark$
- $fmap id \text{ (Just } x\text{)} \equiv \text{ Just } (\text{id } x) \equiv \text{ Just } x \equiv \text{id (Just } x\text{)} \checkmark$

2. La segona llei es pot comprovar també amb els mateixos dos casos:

- Per una banda:  $fmap (g1 . g2) \text{ Nothing} \equiv \text{ Nothing} \equiv \text{id Nothing}$ , per altra banda:  
 $((fmap g1).(fmap g2)) \text{ Nothing} \equiv fmap g1 (fmap g2 \text{ Nothing}) \equiv fmap g1 \text{ Nothing} \equiv \text{ Nothing} \checkmark$

- Per una banda:  $fmap(g1 \cdot g2)(\text{Just } x) \equiv \text{Just}((g1 \cdot g2)x)$  i, per altra banda:  $((fmap g1) \cdot (fmap g2))(\text{Just } x) \equiv fmap g1(\text{Just}(g2 x)) \equiv \text{Just}(g1(g2 x)) \equiv \text{Just}((g1 \cdot g2)x) \checkmark$

b)

```
instance Monad Maybe where
    return = Just
    Nothing  $\gg= f = \text{Nothing}$ 
    Just x  $\gg= f = f x$ 
```

1. La primera llei es pot comprovar així:

- $\text{return } x \gg= f \equiv \text{Just } x \gg= f \equiv f x \checkmark$

2. La segona llei es pot comprovar també amb els dos casos:

- $\text{Nothing} \gg= \text{return} \equiv \text{Nothing} \checkmark$
- $\text{Just } x \gg= \text{return} \equiv \text{return } x \equiv \text{Just } x \checkmark$

3. La tercera llei es pot comprovar també amb els dos casos:

- Per una banda,  $(\text{Nothing} \gg= f) \gg= g \equiv \text{Nothing} \gg= g \equiv \text{Nothing}$ . Per l'altra banda,  $\text{Nothing} \gg= (\lambda x \rightarrow f x \gg= g) \equiv \text{Nothing}$ .  $\checkmark$
- Per una banda,  $(\text{Just } x \gg= f) \gg= g \equiv f x \gg= g$ . Per l'altra banda,  $\text{Just } x \gg= (\lambda x \rightarrow f x \gg= g) \equiv (\lambda x \rightarrow f x \gg= g)x \equiv f x \gg= g$ .  $\checkmark$

### 3. Python

- El programa escriu 0 0 0 seguit de 2 0 2.
- El programa escriu ['unicorn', 'rainbow'].
- c)

```
def nombres(i):
    while True:
        yield i
        i += 1
```

d)

```
def filtre (s, p):
    for x in s:
        if p(x):
            yield x
```

e) Python és un llenguatge amb tipat fort i dinàmic.

### 4. Herència

- La segona i la tercera assignacions són incorrectes.
- A *a1.x2*, la variable *a1* que és de tipus *C1* no té un atribut *x2*.
- En els dos casos es crida al *m* de *C3*.
- La segona i la tercera assignacions són incorrectes.
- A *a1.x2*, la variable *a1* que és de tipus *C1* no té un atribut *x2*.
- En el primer cas es crida al *m* de *C3* i en el segon al de *C*.

## 5. Compilació

```
grammar Gestor;

root      : comandes* EOF;

comandes   : ID ASSIGN expr
            | (LE | WI | ML | MW) param_pas
            ;

expr       : PARO exp PARC
            | expr SQ expr
            | expr PA expr
            | expr AT num
            | MKDIR param_pas
            | MKFILE param_pas
            | mv
            | ID
            ;

param_pas  : PARO ID PARC ;

mv         : MV PARO (mv | expr) COM expr PARC
            ;

LE          : 'length' ;
WI          : 'width' ;
ML          : 'maxlength' ;
MW          : 'maxwidth' ;
MKDIR       : 'mkdir' ;
MKFILE      : 'mkfile' ;
MV          : 'move' ;
ID          : [a-z]+ ;
NUM         : [0-9]+ ;
ASSIGN      : ':=' ;
SQ          : ';' ;
COM         : ',' ;
PA          : '<->' ;
AT          : '^' ;
PARO        : '(' ;
PARC        : ')' ;
WS          : [ \n]+ -> skip ;
```

## 6. Avaluació Única Haskell

```
--a) eval :: Expr -> Mem -> Int

eval (Val x) m = x
eval (Var v) m =
    case Map.lookup 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 = bool2int $ eval' e1 e2 m (/=)
eval (Lth e1 e2) m = bool2int $ eval' e1 e2 m (<)

eval' e1 e2 m op = op (eval e1 m) (eval e2 m)
bool2int = fromEnum -- també es pot més llarg fer amb condicionals

--b) exec :: Instr -> Mem -> Mem

exec (Ass v e) m = Map.insert 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

-- c) exec es penja si el programa es penja

-- d) run :: Instr -> IO ()

run i = mapM_ printVar $ sort $ Map.keys m
where
    m = exec i Map.empty
    printVar k = do
        putStrLn k
        putStrLn " -> "
        print $ fromJust $ Map.lookup k m
```

# Llenguatges de Programació, FIB, 7 de juny de 2019

L'examen dura 3 hores. Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes. No es pot consultar cap material addicional. Feu bona lletra.

Els qui tingueu nota  $\geq 4$  al parcial de Haskell, podeu no fer el darrer problema.

## 1. $\lambda$ -càcul

(2 punts)

Per definir parells (tuples de dos elements) en el  $\lambda$ -càcul, es pot definir una funció  $P$  (per *pair*) que pren dos arguments i retorna la representació d'un parell d'aquells dos valors d'aquesta forma:

$$P \equiv \lambda x. \lambda y. \lambda f. f x y$$

Per exemple, per representar el parell  $(a, b)$ , caldria calcular  $P a b$ , que donaria  $\lambda f. f a b$ .

- Escriviu una  $\lambda$ -expressió  $F$  (per *fst*) que, aplicada sobre una parella representada com s'ha explicat, calculi el seu primer element. Mostreu que  $F(P a b)$  val  $a$  fent explícita cada  $\beta$ -reducció.
- Escriviu una  $\lambda$ -expressió  $S$  (per *snd*) que, aplicada sobre una parella representada com s'ha explicat, calculi el seu segon element. Mostreu que  $S(P a b)$  val  $b$  fent explícita cada  $\beta$ -reducció.

## 2. Sistemes de tipus

(1 punt)

Expliqueu en quina mesura és important que un llenguatge de programació tingui reco-llida de memòria brossa (*garbage collection*, és a dir, un sistema automàtic per alliberar les zones de memòria que ja no són accessibles des del programa) per assegurar seguretat de tipus (*type safety*, és a dir, que no es poden donar errors de tipus en temps d'execució).

## 3. Inferència de tipus

(1'5 punts)

Inferiu (incloent possibles classes) el tipus més general de la funció *after* definida per:

$$\begin{aligned} \textit{after } x [] &= \mathbf{Nothing} \\ \textit{after } x (y:ys) &= \begin{cases} \mathbf{Just } ys \\ | \quad x == y \\ | \quad \mathbf{otherwise} = \textit{after } x ys \end{cases} \end{aligned}$$

Per a fer-ho, dibuixeu l'arbre de sintàxi de les expressions, etiqueteu els nodes amb els seus tipus i genereu les restriccions de tipus. Resoleu-les per obtenir la solució i assenyalieu el resultat final amb un quadre.

#### 4. Python

(1'5 punts)

Recordeu que el mòdul *functools* de Python ofereix una funció **reduce** semblant al **foldl** de Haskell. Aquesta és la seva especificació, copiada de la documentació:

**reduce** (*function*, *sequence* [, *initializer*])

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, **reduce(lambda a, x: a + x, [1, 2, 3, 4, 5])** calculates  $((((1+2)+3)+4)+5)$ . The left argument, *a*, is the accumulated value and the right argument, *x*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and sequence contains only one item, the first item is returned.

- a) Utilitzeu **reduce** per implementar les vostres pròpies versions de **map** i **filter** completant el codi següent (només podeu escriure expressions als llocs dels interrogants):

```
def my_map (function, sequence):
    return reduce(?, ?, ?)

def my_filter ( predicate , sequence ):
    return reduce(?, ?, ?)
```

- b) Quin és el cost asimptòtic en el cas pitjor de *my\_map* i de *my\_filter* aplicats a una seqüència d'*n* elements? Suposeu que el cost de *function* i *predicate* és O(1), i que el cost de **reduce** és O(*n*).

## 5. Haskell

(4 punts)

**Nota:** Els apartats *a*) fins a *h*) d'aquest problema requereixen respostes extremadament curtes. Els apartats *i*) i *j*) es poden respondre amb menys de 10 línies de codi cadascún i puntuen igual que tots els apartats anteriors.

L'acció següent llegeix i retorna una línia de text després d'escriure un prompt *p*:

```
input_with_prompt p = do
    putStrLn (p ++ ":")
    getLine
```

- a)* Quin és el tipus de *input\_with\_prompt*?
- b)* Transformeu *input\_with\_prompt* de notació **do** a notació purament funcional.

La llibreria estàndard de Haskell inclou les funcions **mapM** i **filterM** que són generalitzacions de **map** i **filter** per a mònades. Aquests són els seus tipus:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

Considereu aquest fragment de codi:

```
x = mapM input_with_prompt ["p1", "p2"]
```

- c)* Quin és el tipus de *x*?

Considereu ara aquest altre fragment de codi:

```
y ← mapM input_with_prompt ["p1", "p2"]
```

- d)* Quin és el tipus de *y*?
- e)* Expliqueu quin és l'efecte d'aquest *binding*.

Considereu ara aquest programa:

```
oddity n =
    if odd n then do
        print n
        return True
    else
        return False
```

```
main = filterM oddity [1..10]
```

- f)* Quin és el tipus de *oddity*?
- g)* Expliqueu què fa *oddity*.
- h)* Quina és la sortida del programa principal?
- i)* Doneu una implementació recursiva de **mapM**.
- j)* Doneu una implementació recursiva de **filterM**.

# Llenguatges de Programació, FIB, 7 de junt de 2019

## Possibles solucions

### 1. $\lambda$ -càcul

Utilitzarem dues definicions auxiliars:  $K_1 \equiv \lambda x. \lambda y. x$  i  $K_2 \equiv \lambda x. \lambda y. y$ .  $K_1$  és com el *const* de Haskell: donats dos arguments, retorna el primer, ignorant el segon.  $K_2$  és el complementari: donats dos arguments, retorna el segon, ignorant el primer.

Usarem  $F \equiv \lambda x. x K_1$  i  $S \equiv \lambda x. x K_2$ .

Comprovem que dónen el resultat requerit. Per  $F$ :

$$\begin{aligned} F(\lambda f. fab) &\equiv (\lambda x. x K_1)(\lambda f. fab) \\ &\equiv (\lambda f. fab)K_1 \\ &\equiv K_1 a b \\ &\equiv (\lambda x. \lambda y. x)a b \\ &\equiv (\lambda y. a)b \\ &\equiv a \end{aligned}$$

I per  $S$ :

$$\begin{aligned} S(\lambda f. fa b) &\equiv (\lambda x. x K_2)(\lambda f. fa b) \\ &\equiv (\lambda f. fa b)K_2 \\ &\equiv K_2 a b \\ &\equiv (\lambda x. \lambda y. y)a b \\ &\equiv (\lambda y. y)b \\ &\equiv b \end{aligned}$$

### 2. Sistemes de tipus

Si el programador pot alliberar memòria explícitament, pot tenir punters que no apunten a zones de memòria que corresponen al seu tipus (*dangling pointers*). El resultat és que el LP no és *type-safe*.

Exemple en C++:

```
int* p = new int;
*p = 66;
delete p;
double* q = new double;    // pot ser que p i q apuntin al mateix lloc
*q = 3.1416;              // ok
if (*p == 12) ...          // error de tipus: *p s'usa com a int i no com a double
```

### 3. Inferència de tipus

El tipus final és *after* :: **Eq**  $a \Rightarrow a \rightarrow [a] \rightarrow \text{Maybe } [a]$ .

## 4. Python

a)

```
def my_map (function, sequence):
    return reduce(lambda a, x: a + [f(x)], sequence, [])

def my_filter ( predicate , sequence ):
    return reduce(lambda a, x: a + [x] if predicate (x) else a, sequence, [])
```

b) Els costos són  $O(n^2)$ , ja que afegir un element al final d'una llista amb l'operador + té cost lineal. No és una solució gens eficient!

## 5. Haskell

a) `input_with_prompt :: String → IO String`

b) `input_with_prompt p = putStrLn (p ++ ":") >> getLine`

c) `x :: IO [String]`

d) `y :: [String]`

e) Escriu "p1", llegeix una línia, escriu "p2", llegeix una altra línia i forma una llista amb les dues línies llegides.

f) `oddity :: Int → IO Bool`

g) Donat un enter, `oddity` indica si és senar o no i, si ho és, l'escriu.

h) 1, 3, 5, 7, 9

i)

```
mapM _ [] = return []
mapM f (x:xs) = do
    y ← f x
    ys ← mapM f xs
    return (y:ys)
```

j)

```
filterM _ [] = return []
filterM p (x:xs) = do
    b ← p x
    ys ← filterM p xs
    if b then return (x:ys)
    else return ys
```

# Llenguatges de Programació, FIB, 26 de gener de 2018

L'examen dura 3 hores. Es valorarà la concisió, claredat i brevetat a més de la completenessa i l'exactitud de les respostes. No es pot consultar cap material addicional.

Els qui tingueu nota  $\geq 4$  al parcial de Haskell, podeu no fer els problemes 4 i 5.

## 1. Entrants

(2 punts)

1. **Cultura dels LP.** Quina relació tenien el senyor Haskell (el del llenguatge) i el senyor Curry (el de la *currificació*)?
2.  **$\lambda$ -càcul.** Siguin  $T \equiv \lambda xy.x$  i  $F \equiv \lambda xy.y$  termes per a cert i fals respectivament. Comproveu que  $NEG \equiv \lambda x.(xFT)$  és la negació de booleans tot constraint la seva taula de veritat. Mostreu cada  $\beta$ -reducció aplicada.
3. **Python.** Implementeu una funció  $compose(f, g)$  que, donades dues funcions  $f$  i  $g$  d'un sol paràmetre, retorni una funció que sigui llur composició ( $f \circ g$ ). Per exemple, fent

```
def f(x): return x+1
def g(x): return 2*x
h = compose(f, g)
```

llavors  $h(2)$  hauria de valer 5.

4. **Sistemes de tipus.** Indiqueu les propietats del sistema de tipus del llenguatge que us va tocar en el Treball Dirigit de Competències Transversals (digueu quin era el llenguatge!).

## 2. Inferència de tipus

(2 punts)

Recordeu la funció *fsmap* del parcial? Tant, és... Aquí la teniu:

```
fsmap = foldl $ flip ($)
```

Inferiu el tipus més general de *fsmap* utilitzant els tipus següents:

```
foldl :: (b → a → b) → b → [a] → b
flip  :: (a → b → c) → b → a → c
($)   :: (a → b) → a → b
```

Per a fer-ho, dibuixeu l'arbre decorat de les expressions i genereu les restriccions de tipus. Resoleu-les per obtenir la solució i assenyaleu el resultat final amb un quadre.

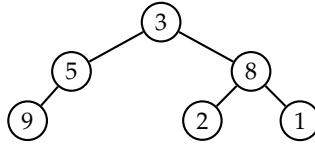
### 3. Arbre general a arbre binari en Python

(2 punts)

Considerem que enmagatzem arbres *binaris* utilitzant nodes formats per tres elements desats en una llista: el valor del node, el subarbre esquerre i el subarbre dret. L'arbre buit es representa amb *None*. Per exemple,

```
[3, [5, [9, None, None], None], [8, [2, None, None], [1, None, None]]]
```

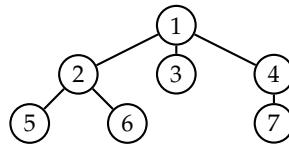
representa aquest arbre binari:



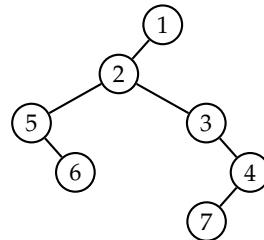
Considerem també que enmagatzem arbres *generals* utilitzant tuples de dos elements: el primer camp és el valor del node, el segon camp és una llista amb els seus fills. No hi ha arbres generals buits, però les fulles tenen una llista buida de fills. Per exemple,

```
(1, [(2, [(5, []), (6, [])]), (3, []), (4, [(7, [])])])
```

representa aquest arbre general:



Es pot convertir un arbre general a un arbre binari equivalent tot utilitzant el mètode de posar el primer fill com a subarbre esquerre i els demés fills com a subarbres cap a la dreta (*left-child right sibling representation*). Per exemple, la transformació a arbre binari de l'arbre general anterior és:



que es representa així:

```
[1, [2, [5, None, [6, None, None]], [3, None, [4, [7, None, None], None]]], None]
```

Feu una funció que, donat un arbre general, el transformi a arbre binari. Es valorarà l'ús d'alguna funció d'ordre superior de Python a la vostra implementació. 10 línies de codi són suficients.

#### 4. Programa misteriós

(2 punts)

Considereu el programa següent en Haskell:

```
import Data.List (sort)

foo = unlines . reverse . take 5 . sort . filter (not . odd . length) . lines

main = do
    bar ← getContents
    let norf = foo bar
    putStrLn norf
```

1. Quin és el tipus de (.) ?
2. Què fa aquest programa?
3. Tranformeu el seu *main* de notació **do** a notació purament funcional.

#### 5. Fluffy i misty

(2 punts)

Definiu les funcions marcades amb punts suspensius amb una implementació no trivial que respecti els tipus.

```
class Fluffy f where
    furry :: (a → b) → f a → f b

instance Fluffy Maybe where
    furry = ...

instance Fluffy [] where
    furry = ...

class Misty m where
    banana :: (a → m b) → m a → m b
    unicorn :: a → m a
    faun    :: (a → b) → m a → m b
    faun = ...
    -- useu banana i/o unicorn

instance Misty Maybe where
    banana = ...
    unicorn = ...

instance Misty [] where
    banana = ...
    unicorn = ...
```

-- Apartat 5.1

-- Apartat 5.2

-- Apartat 5.3

-- Apartat 5.4

-- Apartat 5.5

# Llenguatges de Programació, FIB, 26 de gener de 2018

## Possibles solucions

### 1. Entrants

1. Eren la mateixa persona: Haskell B. Curry (1900–1982).
2. Comprovem que  $\text{NEG } T \equiv F$  i que  $\text{NEG } F \equiv T$ :

$$\begin{cases} \text{NEG } T \equiv \lambda x. (xFT)(\lambda xy.x) \equiv \lambda z. (zFT)(\lambda xy.x) \equiv_{\beta} (\lambda xy.x)(FT) \equiv_{\beta} F \\ \text{NEG } F \equiv \lambda x. (xFT)(\lambda xy.y) \equiv \lambda z. (zFT)(\lambda xy.y) \equiv_{\beta} (\lambda xy.y)(FT) \equiv_{\beta} T \end{cases}$$

3. Així:

```
def compose(f, g):
    return lambda x: f(g(x))
```

4. La solució depèn del vostre llenguatge però no depèn del seu mode d'execució ni paradigma de programació.

### 2. Inferència de tipus

Fent l'arbre decorat, posant les equacions que en surten i resolent-les (no ho faig), s'obté que

$fsmap :: a \rightarrow [a \rightarrow a] \rightarrow a$

Errors comuns:

- No posar el primer \$ a l'arbre: El sistema d'inferència de tipus no pot donar-li la semàntica d'obrir parèntesi. \$ és una funció com qualsevol altra.
- No rebatejar les variables de tipus que apareixen a les fulles de l'arbre.

### 3. Arbre general a arbre binari en Python

```
def gal2bin(t):
    x, children = t
    if children == []:
        return [x, None, None]
    else:
        roots = map(gal2bin, children)
        for i in range(len(roots) - 1):
            roots[i][2] = roots[i+1]
        return [x, roots[0], None]
```

## 4. Programa misteriós

1. El tipus de l'operador de composició és

(.) ::  $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

2. Donada una seqüència de línies, el programa escriu, les cinc (o menys, si n'ho hi ha prou) línies més petites en ordre lexicogràfic decreixent que tenen longitud parell.

3. La transformació és:

`main = getContents >>= \bar{b}ar \rightarrow putStrLn (foo bar)`

## 5. Fluffy i misty

`furry f Nothing = Nothing` -- 5.1  
`furry f (Just x) = Just (f x)`

`furry = map` -- 5.2

`faun f = banana (unicorn . f)` -- 5.3

`banana f Nothing = Nothing` -- 5.4  
`banana f (Just x) = f x`  
`unicorn = Just`

`banana f xs = concat $ map f xs` -- 5.5  
`unicorn x = [x]`

[Problema extret de *20 Intermediate Haskell Exercises* de Tony Morris.]