Llenguatges de Programació

Sessió 3: llistes infinites

Jordi Petit, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Contingut

Llistes amb rangs

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

... amb salt

```
λ> [10, 20 .. 100]
👉 [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
λ> [10, 9 .. 1]
👉 [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

λ> [1, 2, 4, 8, 16 .. 256]
❌ -- no fa miracles
```

... sense final

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

Llistes per comprensió

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

Sintaxi semblant a la notació matemàtica de construcció de conjunts.

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

Ternes pitagòriques en Haskell (fins a `n`):

```
λ> ternes n = [(x, y, z) | x <- [1..n],
                           y <- [x..n],
                           z <- [y..n], x*x + y*y == z*z]
    -- gens eficient

λ> ternes 20
👉 [(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

Llistes per comprensió

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

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

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

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

Múltiples filtres

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

Múltiples generadors (producte cartesià)

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

Introducció de noms

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

Llistes per comprensió

Compte amb l'ordre

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

Ternes pitagòriques

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

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

Perspectiva

Haskell

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

Python

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

SQL

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

C++

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

Contingut

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

Avaluació mandrosa

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

Avaluació mandrosa: C++ vs Haskell

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

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

💣: Divisió per zero quan `b` és zero.

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

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

💣: Es penja.

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

👍 `?:`, `and` i `or` sí són mandroses.

```
λ> f x y = x
λ> a = 2
λ> b = 0
λ> f a (div a b)
👉 2
```

👍 `(div a b)` no és avaluat.

```
λ> f x y = x
λ> h x = h x
λ> f 3 (h 0)
👉 3
```

👍 `h` mai és avaluada.

Visualització dels *thunks* amb el *ghci*

```
λ> xs = [x + 1 | x <- [1..10]] :: [Int]

λ> :sprint xs
xs = _

λ> null xs
False

λ> :sprint xs
xs = _ : _

λ> head xs
2

λ> :sprint xs
xs = 2 : _

λ> length xs
10

λ> :sprint xs
xs = [2,_,_,_,_,_,_,_,_,_]
```

```
λ> y = head $ tail $ tail xs

λ> :sprint y
y = _

λ> :sprint xs
xs = [2,_,_,_,_,_,_,_,_,_]

λ> y
4

λ> :sprint y
y = 4

λ> :sprint xs
xs = [2,_,4,_,_,_,_,_,_,_]

λ> xs
[2,3,4,5,6,7,8,9,10,11]

λ> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Avaluació mandrosa pas a pas

Donades les definicions (`nats` amb recursivitat infinita):

```
nats = 0 : map (+1) nats          (N)

map f (x:xs) = f x : map f xs     (M)
```

```
take 0 _ = []                     (T1)
take n (x:xs) = x : take (n-1) xs (T2)
```

Avaluació pas a pas de `take 2 nats`:

expressió	regla
`take 2 nats`	
`take 2 (0 : map (+1) nats)`	(N)
`0 : take 1 (map (+1) nats)`	(T2)
`0 : take 1 (map (+1) (0 : map (+1) nats))`	(N)
`0 : take 1 ((+1) 0 : (map (+1) (map (+1) nats)))`	(M)
`0 : take 1 (1 : (map (+1) (map (+1) nats)))`	(+1)
`0 : 1 : take 0 (map (+1) (map (+1) nats))`	(M)
`0 : 1 : []`	(T1)

Contingut

Zeros

Generació de la llista infinita de zeros

```
zeros :: [Int]

-- amb repeat
zeros = repeat 0

-- amb cycle
zeros = cycle [0]

-- amb iterate
zeros = iterate id 0

-- amb recursivitat infinita
zeros = 0 : zeros

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

Naturals

Generació de la llista infinita de naturals

```
naturals :: [Integer]

-- amb rangs infinits
naturals = [0..]

-- amb iterate
naturals = iterate (+1) 0

-- amb recursivitat infinita
naturals = 0 : map (+1) naturals

-- prova
λ> take 6 naturals
👉 [0, 1, 2, 3, 4, 5]
```

Factorials

Generació de la llista infinita de factorials

```
factorials :: [Integer]

factorials = 1:zipWith (*) factorials [1..]

λ> take 10 factorials
👉 [1,1,2,6,24,120,720,5040,40320,362880]
```

```
factorials :: [Integer]

factorials = scanl (*) 1 [1..]

λ> take 6 $ scanl (*) 1 [1..]
👉 [1, 1, 2, 6, 24, 120]
```

Fibonacci

Generació de la llista infinita de nombres de Fibonacci

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

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

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

Primers

Generació dels nombres primers amb el Garbell d'Eratòstenes

```haskell
primers :: [Integer]

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

Avaluació ansiosa

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

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

Contingut

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

Exercicis

Feu aquests problemes de Jutge.org:

- P93588 Usage of comprehension lists
- P98957 Infinite lists