Llenguatges de Programació

Sessió 2: funcions d'ordre superior

Jordi Petit

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Contingut

Funcions d'ordre superior

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

Punt clau: les funcions són objectes de primera classe.

Exemple en C++:

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

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

Funcions d'ordre superior

Exemples:

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

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

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

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

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

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

Funcions d'ordre superior

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

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

```
λ> apli2 sqrt 16.0
👉 2.0
```

De forma equivalent:

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

```
λ> apli2 sqrt 16.0
👉 2.0
```

Petit exercici:

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

Funcions anònimes

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

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

Funció amb nom:

```
doble x = 2 * x                    -- equival a doble = \x -> 2 * x

λ> doble 3                         👉 6
λ> map doble [1, 2, 3]             👉 [2, 4, 6]
```

Funció anònima:

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

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

També són útils per realitzar transformacions de programes.

Funcions anònimes

Múltiples paràmetres:

```
\x y -> x + y
```

és equivalent a

```
\x -> \y -> x + y
```

que vol dir

```
\x -> (\y -> x + y)
```

Seccions

Les **seccions** permeten aplicar operadors infixos parcialment.

Per la dreta:
`(⊗ y) ≡ \x -> x ⊗ y`

Per l'esquerra:
`(y ⊗) ≡ \x -> y ⊗ x`

Exemples:

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

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

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

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

Contingut

Funcions d'ordre superior habituals

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

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

composició `(.)`

- Signatura:

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

- Descripció:

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

- Exemples:

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

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

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

aplicació `($)`

- Signatura:

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

- Descripció:

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

- Exemples:

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

const

- Signatura:

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

- Descripció:

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

- Exemples:

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

id

- Signatura:

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

- Descripció:

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

- Exemples:

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

flip

- Signatura:

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

- Descripció:

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

```
flip f x y = f y x
```

- Exemples:

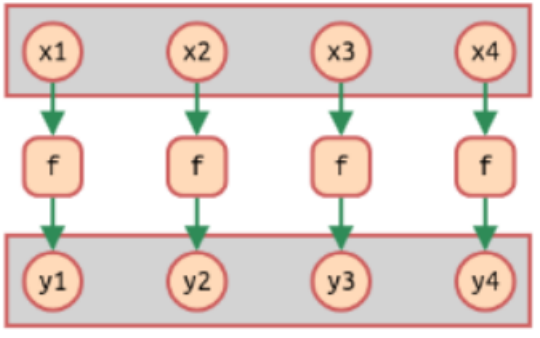
```
λ> meitat = flip div 2

λ> meitat 10
👉 5
```

map

- Signatura:

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

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



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

- Exemples:

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

filter

- Signatura:

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

- Descripció:

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

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

- Exemples:

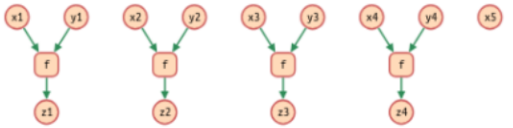
```
λ> filter even [2, 1, 4, 6, 7]
👉 [2, 4, 6]
```

zipWith

- Signatura:

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

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



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

- Exemples:

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

all

- Signatura:

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

- Descripció:

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

- Exemples:

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

any

- Signatura:

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

- Descripció:

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

- Exemples:

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

dropWhile

- Signatura:

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

- Descripció:

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

- Exemples:

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

takeWhile

- Signatura:

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

- Descripció:

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

- Exemples:

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

iterate

- Signatura:

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

- Descripció:

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

  `ys = iterate f x`

- Exemples:

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

until

- Signatura:

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

- Descripció: `until p f x` calcula la llista `[x, f x, f (f x), f (f (f x)), ...]` i retorna el primer element que satisfa el predicat `p`.

- Exemples:

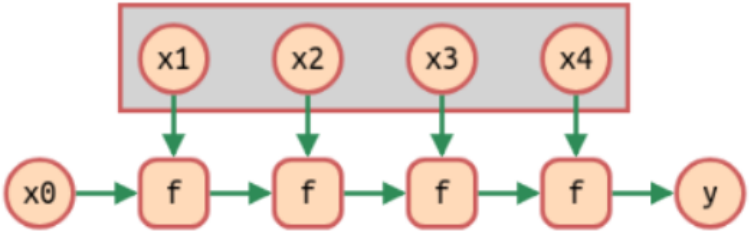
```
λ> until (>100) (*3) 1
👉 243
```

foldl

- Signatura:

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

- Descripció: `foldl ⊕ x0 xs` desplega un operador ⊕ per l'esquerra, de forma que `foldl ⊕ x0 [x1, x2, ..., xn]` és `(((x0 ⊕ x1) ⊕ x2) ⊕ ...) ⊕ xn`.



`y = foldl f x0 [x1, x2, x3, x4]`

- Exemples:

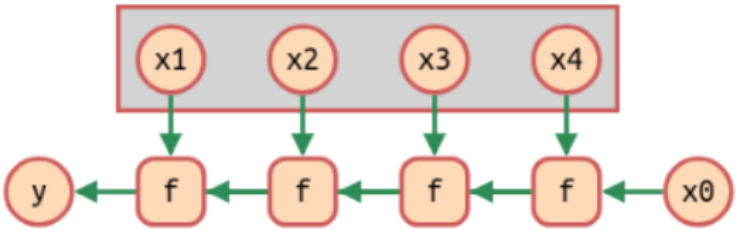
```
λ> foldl (+) 0 [3, 2, (-1)]              -- (((0 + 3) + 2) + (-1))
👉 4
```

foldr

- Signatura:

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

- Descripció: `foldr ⊕ x0 xs` desplega un operador per la dreta, de forma que `foldr ⊕ x0 [x1, x2, ..., xn]` és `x1 ⊕ (x2 ... ⊕ (xn ⊕ x0))`.



`y = foldr f x0 [x1, x2, x3, x4]`

- Exemples:

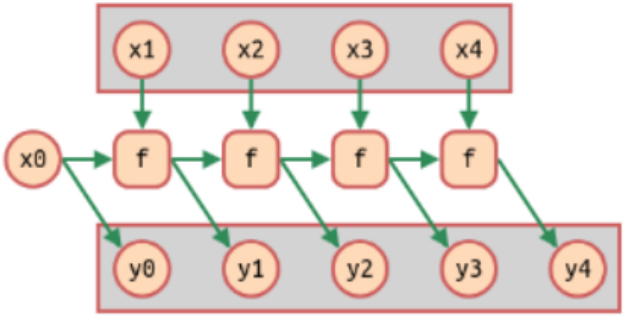
```
λ> foldr (+) 0 [3, 2, (-1)]              -- 3 + ((2 + ((-1) + 0)))
👉 4
```

scanl

- Signatura:

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

- Descripció: `scanl f x0 xs` és com `foldl f x0 xs` però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.



`[y0, y1, y2, y3, y4] = scanl f x0 [x1, x2, x3, x4]`

- Exemples:

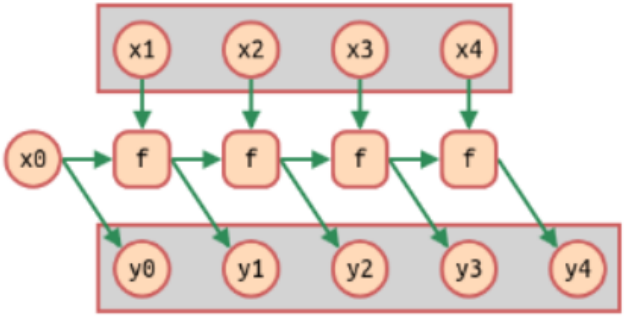
```
λ> scanl (+) 0 [3, 2, (-1)]
👉 [0, 3, 5, 4]
```

scanr

- Signatura:

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

- Descripció: `scanr f x0 xs` és com `foldr f x0 xs` però enlloc de retornar el valor final, retorna la llista amb tots els resultats intermigs.



`[y0, y1, y2, y3, y4] = scanr f x0 [x1, x2, x3, x4]`

- Exemples:

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

Perspectiva

map

- C++

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

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

- Haskell

```
ys = map func xs
```

Perspectiva

filter

- C++

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

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

- Haskell

```
ys = filter pred xs
```

Perspectiva

foldl

- C++

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

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

- Haskell

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

Perspectiva

composició

- Haskell

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

- Shell

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

Contingut

Diccionaris

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

Interfície

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

create :: Int -> Dict
search :: Dict -> String -> Int
insert :: Dict -> String -> Int -> Dict
```

Primera versió

```
type Dict = (String -> Int)

create def = \key -> def

search dict key = dict key

insert dict key value = \x ->
    if key == x then value
    else search dict x
```

Segona versió

```
type Dict = (String -> Int)

create = const

search = ($)

insert dict key value x
    | key == x      = value
    | otherwise     = dict x
```

Dividir i vèncer

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

Interfície

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

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

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

Dividir i vèncer

Solució

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

dIv trivial directe dividir vènçer x
    | trivial x     = directe x
    | otherwise     = vènçer x (x1, x2) (y1, y2)
                          where
                              (x1, x2) = dividir x
                              y1 = dIv trivial directe dividir vènçer x1
                              y2 = dIv trivial directe dividir vènçer x2
```

Dividir i vèncer

Solució capturant el context

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

dIv trivial directe dividir vènçer = dIv'
    where
        dIv' x
            | trivial x = directe x
            | otherwise = vènçer x (x1, x2) (y1, y2)
                                where
                                    (x1, x2) = dividir x
                                    y1 = dIv' x1
                                    y2 = dIv' x2
```

Dividir i vèncer

Implementació de Quicksort amb Dividir i vèncer

```
qs :: Ord a => [a] -> [a]

qs = dIv trivial directe dividir vèncer
  where
    trivial []   = True
    trivial [_]  = True
    trivial _    = False

    directe = id

    dividir (x:xs) = (menors, majors)
      where menors = filter (<= x) xs
        majors = filter (>  x) xs

    dividir' (x:xs) = partition (<= x) xs   -- equivalent amb funció predefinida

    vèncer (x:_) _ (ys1, ys2) = ys1 ++ [x] ++ ys2
```

Contingut

Exercicis

1. Feu aquests problemes de Jutge.org:
   - P93632 Usage of higher-order functions (1)
   - P31745 Usage of higher order functions (2)
   - P90677 Definition of higher-order functions (1)
   - P71775 Definition of higher-order functions (2)

2. Re-implementeu les funcions habituals sobre llistes.
   - Useu `myLength` enlloc de `length` per evitar xocs de noms.
   - No useu recursivitat: useu funcions d'ordre superior.

3. Busqueu a Hoogλe informació sobre aquestes funcions:
   - `foldl1`, `foldr1`, `scanl1`, `scanr1`
   - `partition`
   - `concatMap`
   - `zipWith3`
   - `mapAccumL`, `mapAccumR`