Programmation fonctionnelle et Haskell for dummies

INFO1 - Semaine 41

Guillaume CONNAN

IUT de Nantes - Dpt d'informatique

Dernière mise à jour : 8 octobre 2013 à 12:09

Sommaire

Sommaire

En C

```c
#include <stdio.h>

int somme(int,int);

void main(void)
{ int s,a,b;
  printf("Entrez la valeur du plus petit entier: ");
  scanf("%d" , &a);
  printf("Entrez la valeur du plus grand entier: ");
  scanf("%d" , &b);
  s = somme(a,b);
  printf("La somme des entiers de  %d à %d est %d \n", a,b,s);
}
int somme(int a, int b)
{ int tmp; int som;
  tmp = a;
  som = a;
  while (tmp != b)
   { tmp = tmp + 1;
     som = som + tmp;}
  return som;
}
```

```
$ gcc -o somme poly_haskell.c
$ ./somme
Entrez la valeur du plus petit entier: 1
Entrez la valeur du plus grand entier: 10
La somme des entiers de  1 à 10 est 55
```

Que s'est-il passé ?

A t-on ce que l'on désirait ?

```
$ gcc -o somme poly_haskell.c
$ ./somme
Entrez la valeur du plus petit entier: 1
Entrez la valeur du plus grand entier: 10
La somme des entiers de  1 à 10 est 55
```

Que s'est-il passé ?

A t-on ce que l'on désirait ?

```
$ gcc -o somme poly_haskell.c
$ ./somme
Entrez la valeur du plus petit entier: 1
Entrez la valeur du plus grand entier: 10
La somme des entiers de  1 à 10 est 55
```

Que s'est-il passé ?

A t-on ce que l'on désirait ?

Avec Haskell

```
somme a b = foldl (+) 0 [a..b]
```

```
*Main> somme 1 10
55
```

```
*Main> sum [1..10]
55
```

Avec Haskell

```
somme a b = foldl (+) 0 [a..b]
```

```
*Main> somme 1 10
55
```

```
*Main> sum [1..10]
55
```

Avec Haskell

```
somme a b = foldl (+) 0 [a..b]
```

```
*Main> somme 1 10
55
```

```
*Main> sum [1..10]
55
```

```
somme_fonction f a b = foldl (\accu x -> accu + f(x)) 0 [a..b]
```

```
*Main> somme_fonction (\x -> x^2) 1 10
385
```

```
somme_fonction f a b = foldl (\accu x -> accu + f(x)) 0 [a..b]
```

```
*Main> somme_fonction (\x -> x^2) 1 10
385
```

Calculez la somme des carrés impairs inférieurs à 1000 ?

En C ?

```
sum (takeWhile (<1000) (filter odd (map (^2) [1..])))
```

Calculez la somme des carrés impairs inférieurs à 1000 ?
En C ?

```
sum (takeWhile (<1000) (filter odd (map (^2) [1..])))
```

Calculez la somme des carrés impairs inférieurs à 1000 ?
En C ?

```
sum (takeWhile (<1000) (filter odd (map (^2) [1..])))
```

John Backus (1924 - 2007)

Much of my work has come from being lazy

John Backus (1924 - 2007)

Much of my work has come from being lazy

Sommaire

Spécifier

SPÉCIFIER la fonction : c'est-à-dire parfois (on peut effectivement s'en passer au besoin) la nommer, donner son *type*, i.e. son domaine et son codomaine : on parle de *signature de type* et on explique ce qu'elle fait.

```
double :: Int -> Int
-- calcule le double d'un entier : le résultat est un entier
```

Spécifier

SPÉCIFIER la fonction : c'est-à-dire parfois (on peut effectivement s'en passer au besoin) la nommer, donner son *type*, i.e. son domaine et son codomaine : on parle de *signature de type* et on explique ce qu'elle fait.

```
double :: Int -> Int
-- calcule le double d'un entier : le résultat est un entier
```

Réaliser

RÉALISER la fonction c'est associer à la fonction spécifiée une expression. Pour cela, on utilise des *paramètres formels* qui font *abstraction* des valeurs particulières qui leur seront ensuite *substituées*.

```
double n = 2 * n
```

Réaliser

RÉALISER la fonction c'est associer à la fonction spécifiée une expression. Pour cela, on utilise des *paramètres formels* qui font *abstraction* des valeurs particulières qui leur seront ensuite *substituées*.

```
double n = 2 * n
```

Utiliser

UTILISER a fonction dans une expression en lui donnant des *paramètres effectifs* (arguments), c'est-à-dire liés à l'application particulière de la fonction :

```
*Main> (double 3) + (double 7)
20
```

Utiliser

UTILISER a fonction dans une expression en lui donnant des *paramètres effectifs* (arguments), c'est-à-dire liés à l'application particulière de la fonction :

```
*Main> (double 3) + (double 7)
20
```

Sommaire

Haskell Curry (1900 - 1982)


KEEP
CALM
AND
CURRY
ON

Une **fonction curryfiée** est une fonction de plusieurs variables transformée en une fonction d'une seule variable qui renvoie une fonction ayant une variable de moins...

Cela permet de créer des fonctions partielles.

Une **fonction curryfiée** est une fonction de plusieurs variables transformée en une fonction d'une seule variable qui renvoie une fonction ayant une variable de moins...
Cela permet de créer des fonctions partielles.

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y        =  x + y
```

Associativité

```
f = plus 3
```

Que vaut f(5) ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut f(5) ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut f(5) ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut f(5) ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut `f(5)` ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut `f(5)` ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

```
plus x y = x + y
```

```
plus :: Integer -> Integer -> Integer
plus    x          y       =  x + y
```

Associativité

```
f = plus 3
```

Que vaut `f(5)` ?

```
*Main> f 5
8
```

```
*Main> :t f
f :: Integer -> Integer
```

$\mathcal{F}(D, A)$

$\texttt{plus} \in \mathcal{F}(\mathbb{N}, \mathcal{F}(\mathbb{N}, \mathbb{N}))$

$\mathcal{F}(D, A)$
$\texttt{plus} \in \mathcal{F}(\mathbb{N}, \mathcal{F}(\mathbb{N}, \mathbb{N}))$

Sommaire

Exemple 1

Décrire une fonction qui étant donnés quatre flottants leur associe la moyenne des deux nombres parmi les quatre qui ne sont ni le plus grand ni le plus petit. Par exemple, la moyenne olympique de 10, 8, 12 et 14 est 11 et celle de 12, 12, 12 et 12 est 12.

Spécification

```
moyenne_olympique4 :: Float -> Float -> Float -> Float -> Float
-- renvoie la moyenne olympique de 4 flottants sous forme d'un
    flottant
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

```
*Main> :t min
min :: Ord a => a -> a -> a
```

```
*Main> min 'a' 'b'
'a'
*Main> min 5 3
3
*Main> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
*Main> min True False
False
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

```
*Main> :t min
min :: Ord a => a -> a -> a
```

```
*Main> min 'a' 'b'
'a'
*Main> min 5 3
3
*Main> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
*Main> min True False
False
```

Réalisation

Nous allons par exemple additionner les quatre nombres, retirer le plus grand et le plus petit puis diviser par 2.

```
*Main> :t min
min :: Ord a => a -> a -> a
```

```
*Main> min 'a' 'b'
'a'
*Main> min 5 3
3
*Main> min "Tralala" "Pouet Pouet"
"Pouet Pouet"
*Main> min True False
False
```

Réalisation

```
moyenne_olympique4 a b c d =
  (s - mini - maxi) / 2 -- l'expression correspondant à la méthode
      choisie
  where s = a + b + c + d -- la somme des 4 nombres
        mini = min a (min  b (min c d) ) -- le plus petit des 4
        maxi = max a (max  b (max c d) ) -- le plus grand des 4
```

```
*Main> moyenne_olympique4 10 8 12 14
11.0
```

Réalisation

```
moyenne_olympique4 a b c d =
  (s - mini - maxi) / 2 -- l'expression correspondant à la méthode
      choisie
  where s = a + b + c + d -- la somme des 4 nombres
        mini = min a (min  b (min c d) ) -- le plus petit des 4
        maxi = max a (max  b (max c d) ) -- le plus grand des 4
```

```
*Main> moyenne_olympique4 10 8 12 14
11.0
```

Sommaire

Exemple 2

Décrire une fonction qui étant donnée une liste d'au moins quatre nombres leur associe la moyenne des nombres parmi ceux de la liste qui ne sont ni le plus grand ni le plus petit. Par exemple, la moyenne olympique de 15, 7, 10, 8, 12 et 14 est 11.

Min et Max polymorphes

```
min_liste :: (Ord a) => [a] -> a
-- renvoie le mini d'une liste d'éléments ordonnables
```

```
max_liste :: (Ord a) => [a] -> a
-- renvoie le maxi d'une liste d'éléments ordonnables
```

```
extr_liste :: (Ord a) => (a -> a -> a) -> [a] -> a
-- renvoie le mini ou le maxi (on choisit en mettant la nature de l'
    extremum en paramètre) d'une liste d'éléments ordonnables
```

Min et Max polymorphes

```
min_liste :: (Ord a) => [a] -> a
-- renvoie le mini d'une liste d'éléments ordonnables
```

```
max_liste :: (Ord a) => [a] -> a
-- renvoie le maxi d'une liste d'éléments ordonnables
```

```
extr_liste :: (Ord a) => (a -> a -> a) -> [a] -> a
-- renvoie le mini ou le maxi (on choisit en mettant la nature de l'
    extremum en paramètre) d'une liste d'éléments ordonnables
```

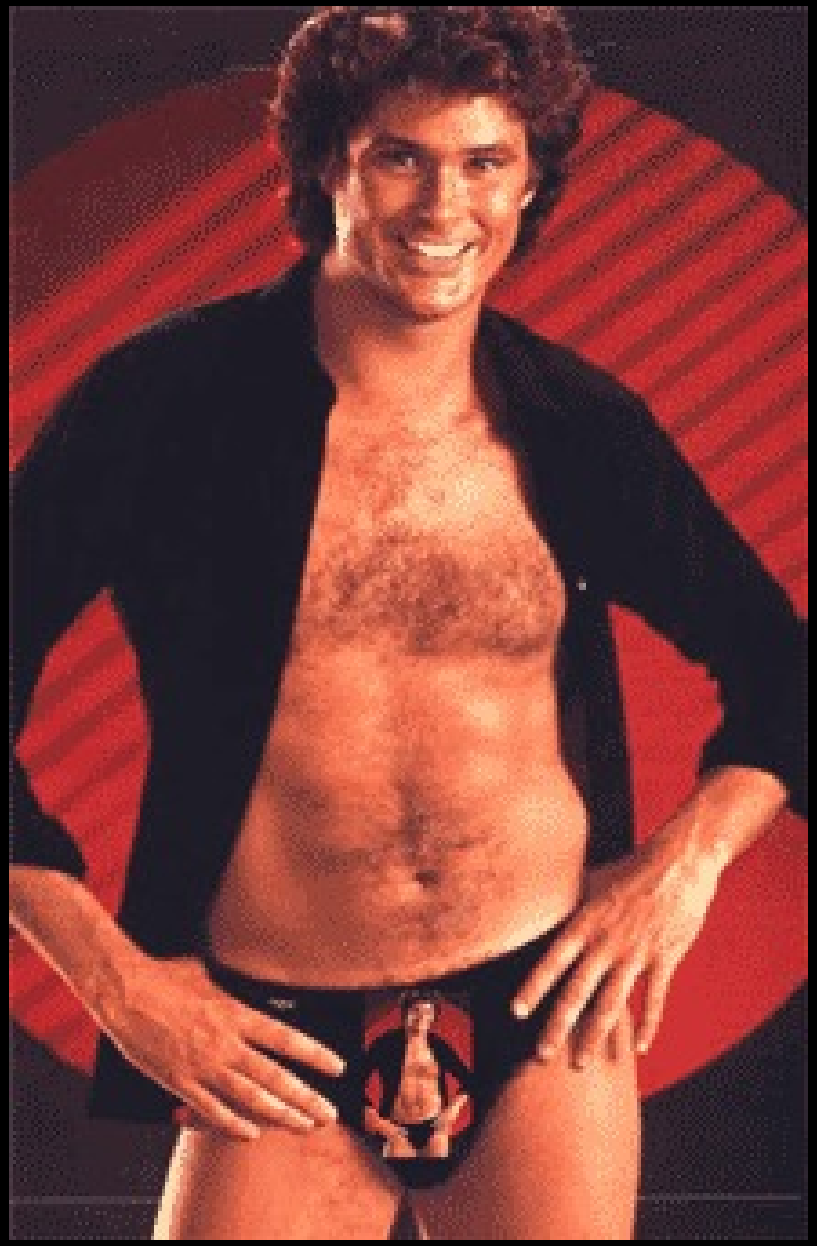
Min et Max polymorphes

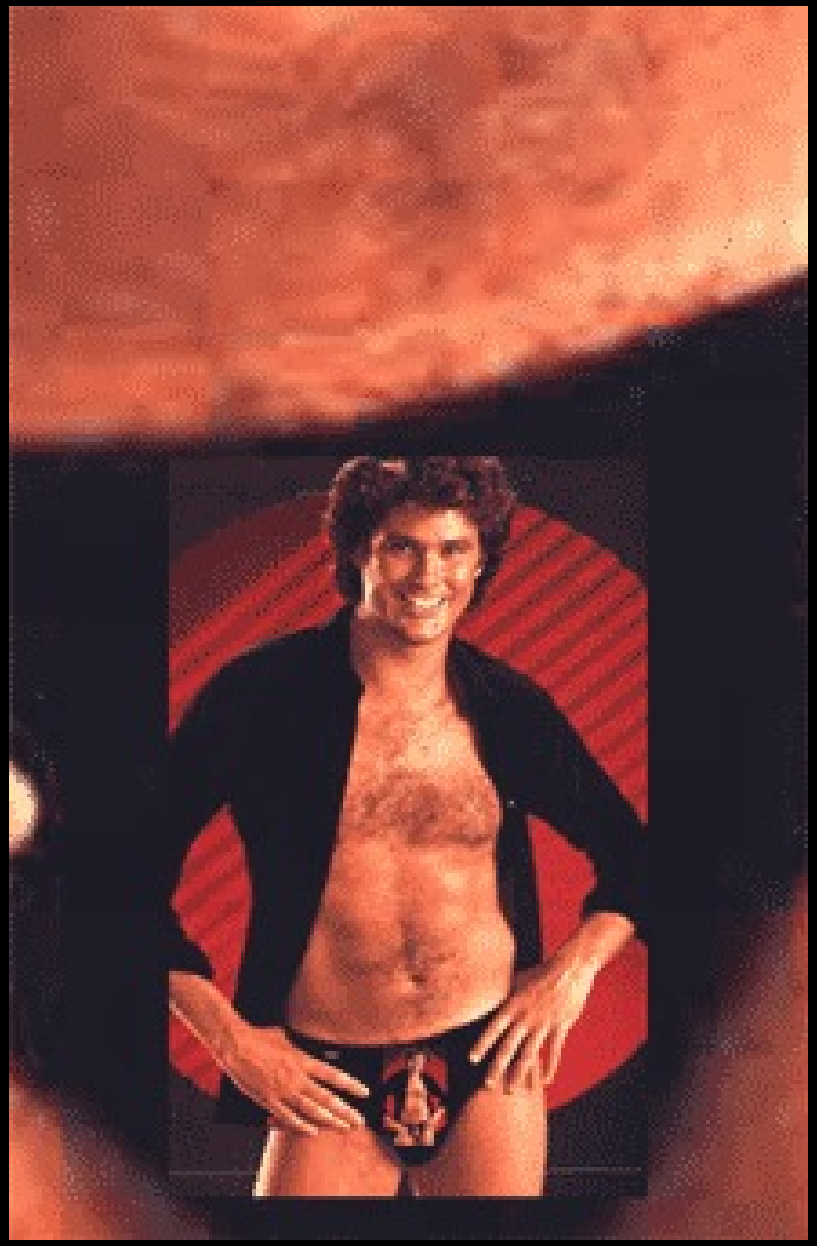
```
min_liste :: (Ord a) => [a] -> a
-- renvoie le mini d'une liste d'éléments ordonnables
```

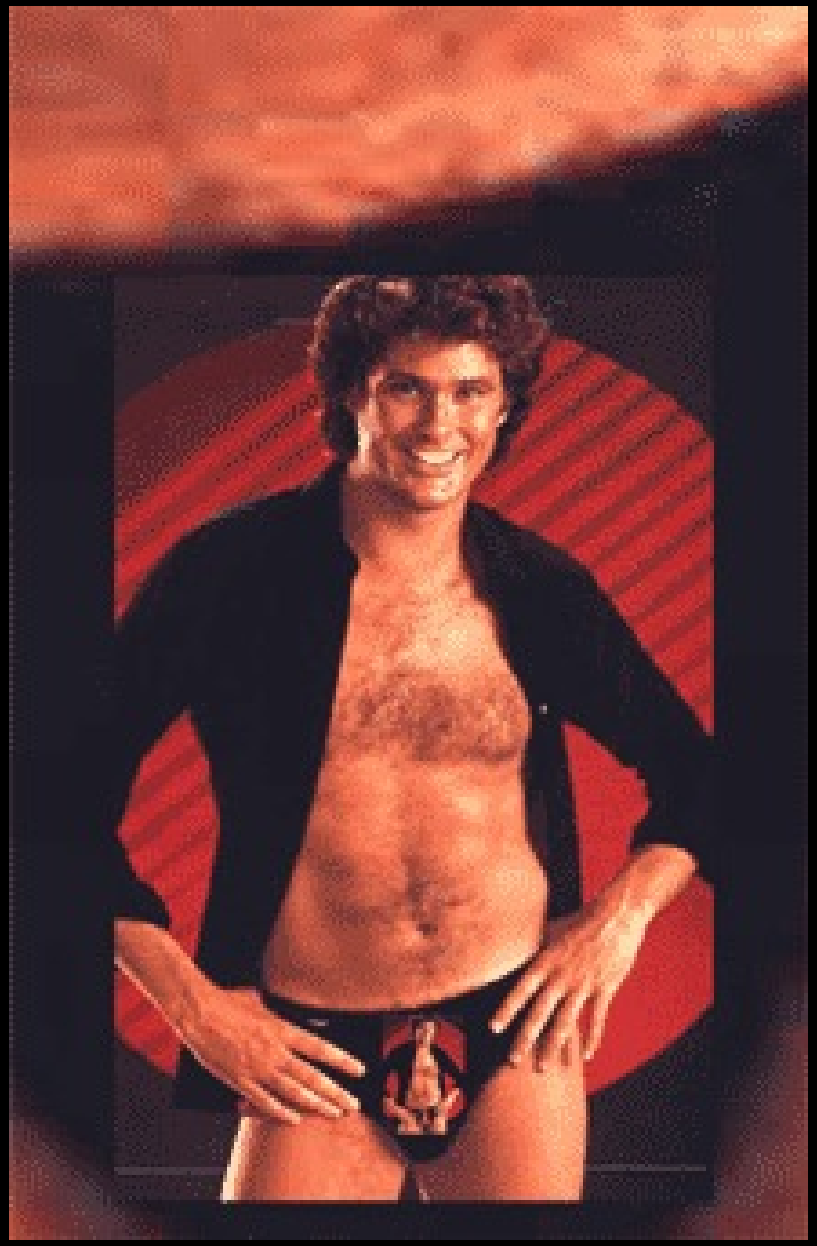
```
max_liste :: (Ord a) => [a] -> a
-- renvoie le maxi d'une liste d'éléments ordonnables
```

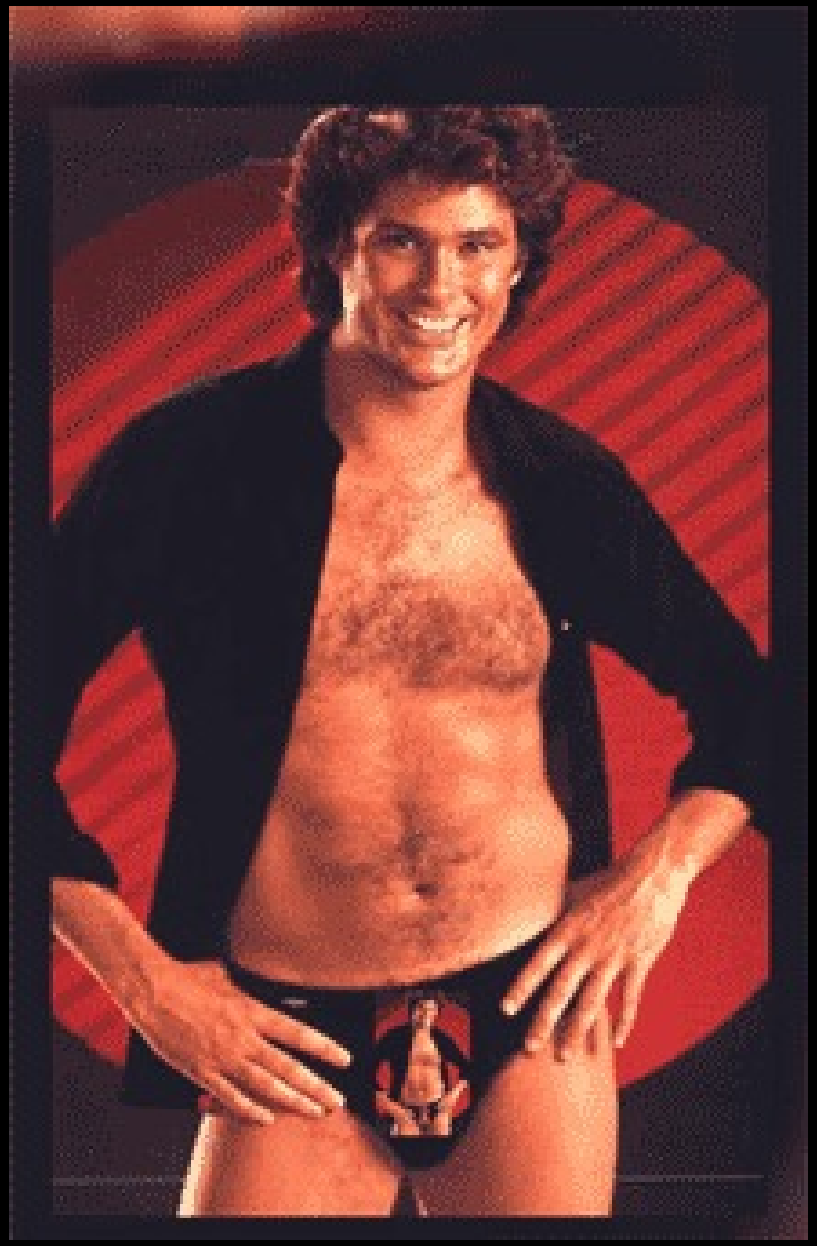
```
extr_liste :: (Ord a) => (a -> a -> a) -> [a] -> a
-- renvoie le mini ou le maxi (on choisit en mettant la nature de l'
    extremum en paramètre) d'une liste d'éléments ordonnables
```

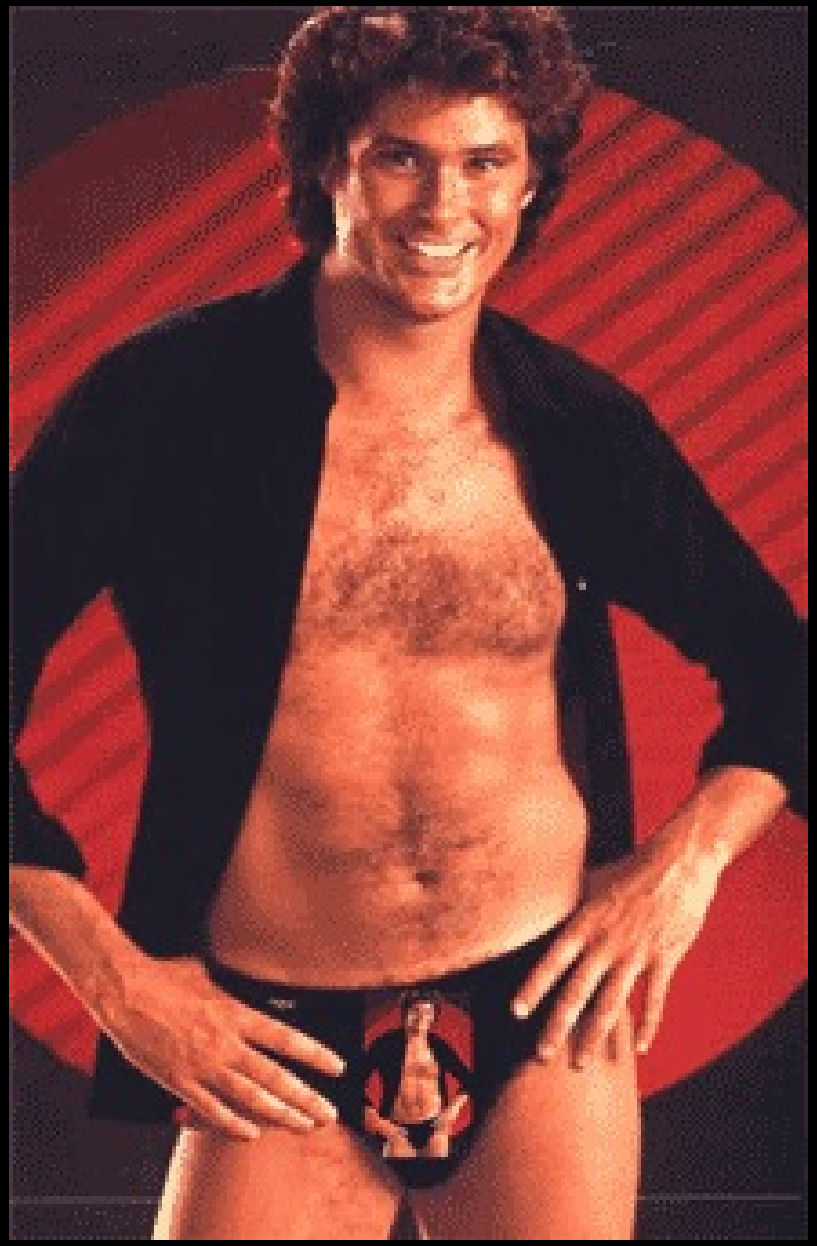

La Vache qui rit

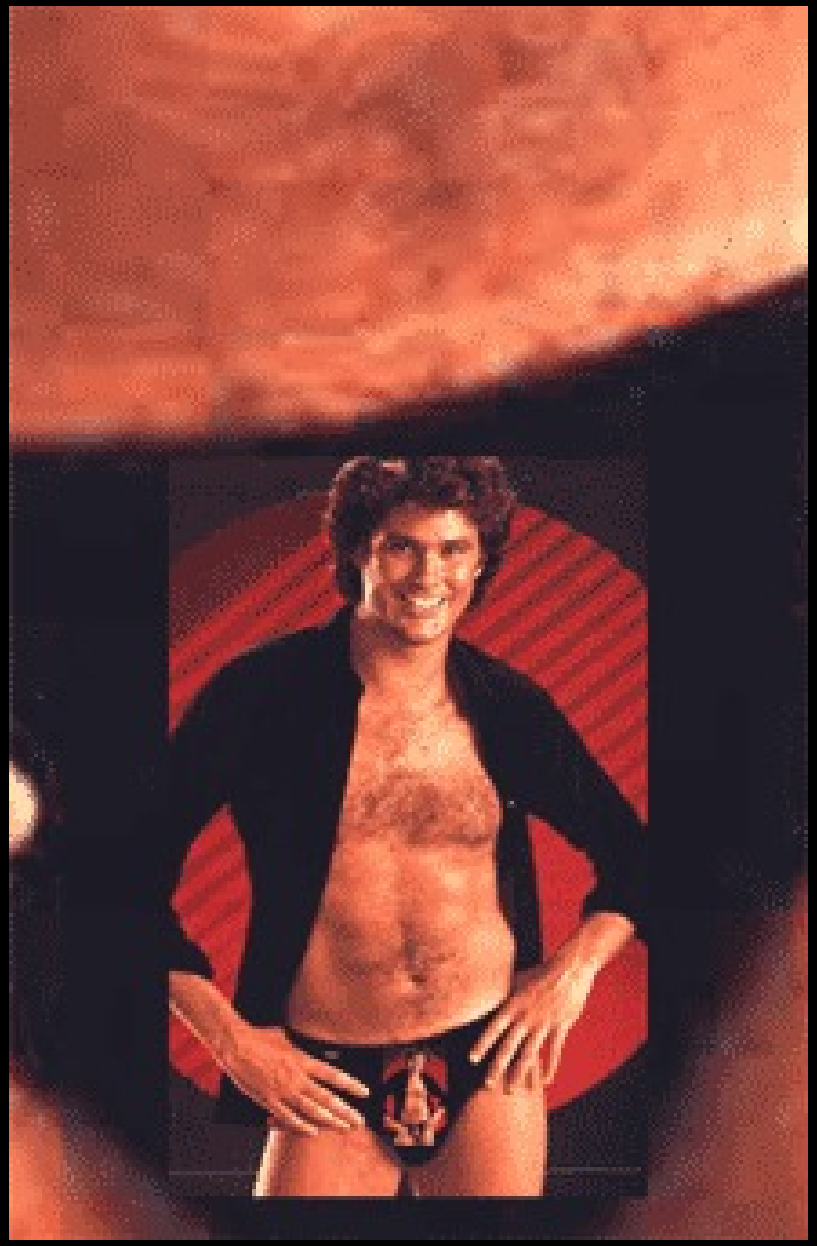


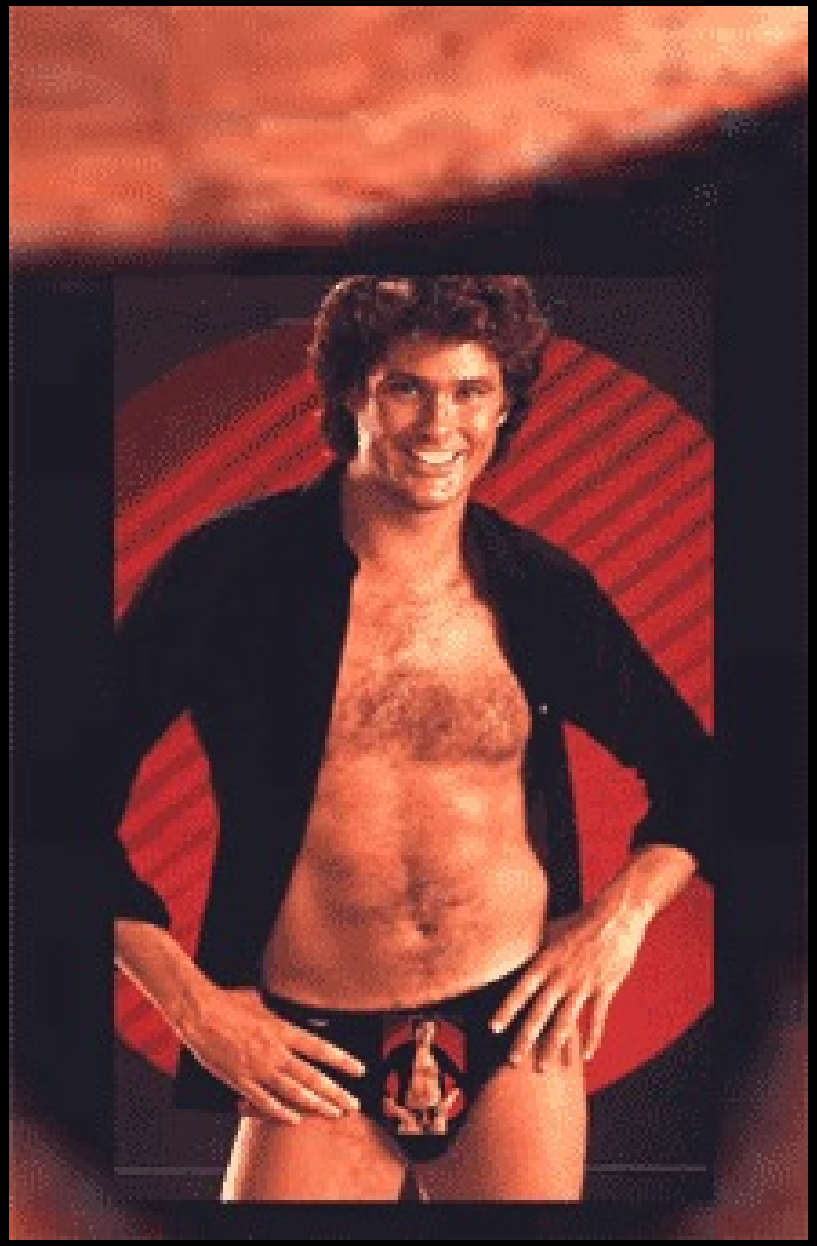


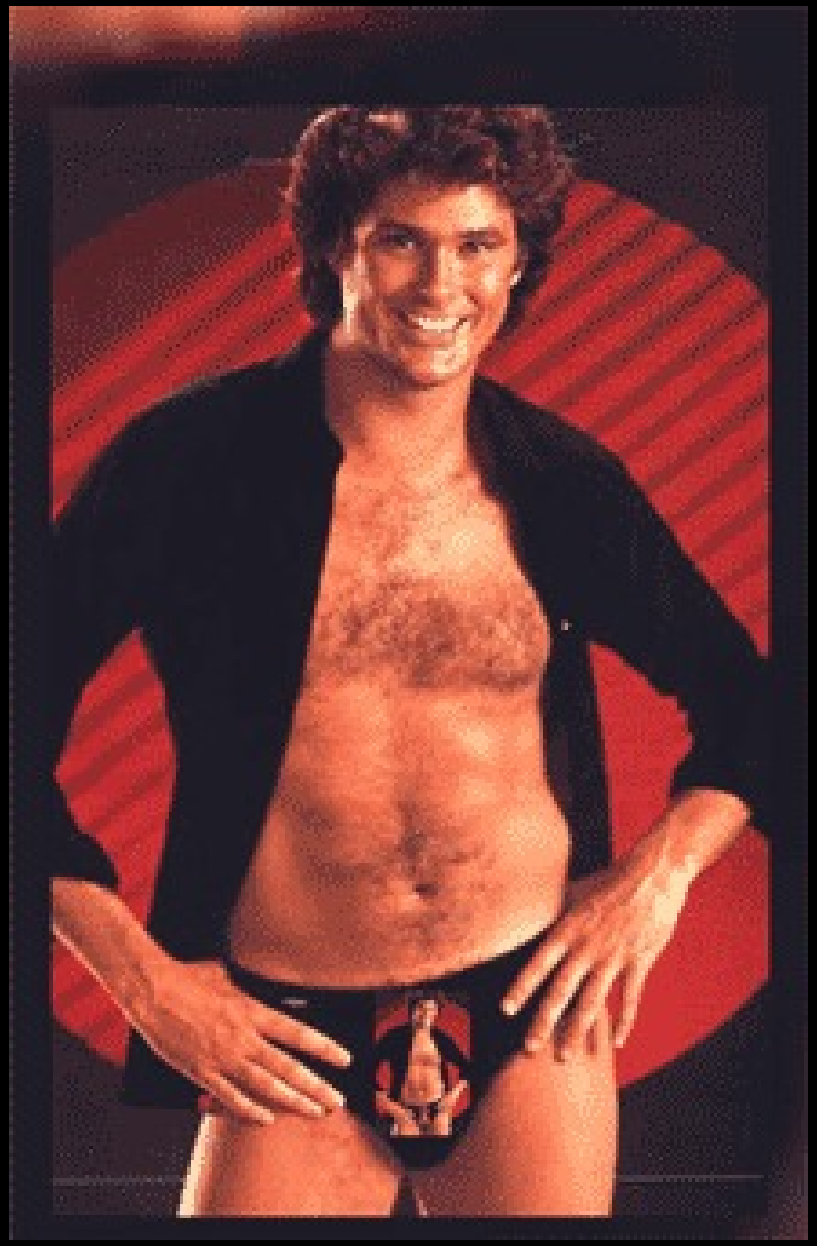


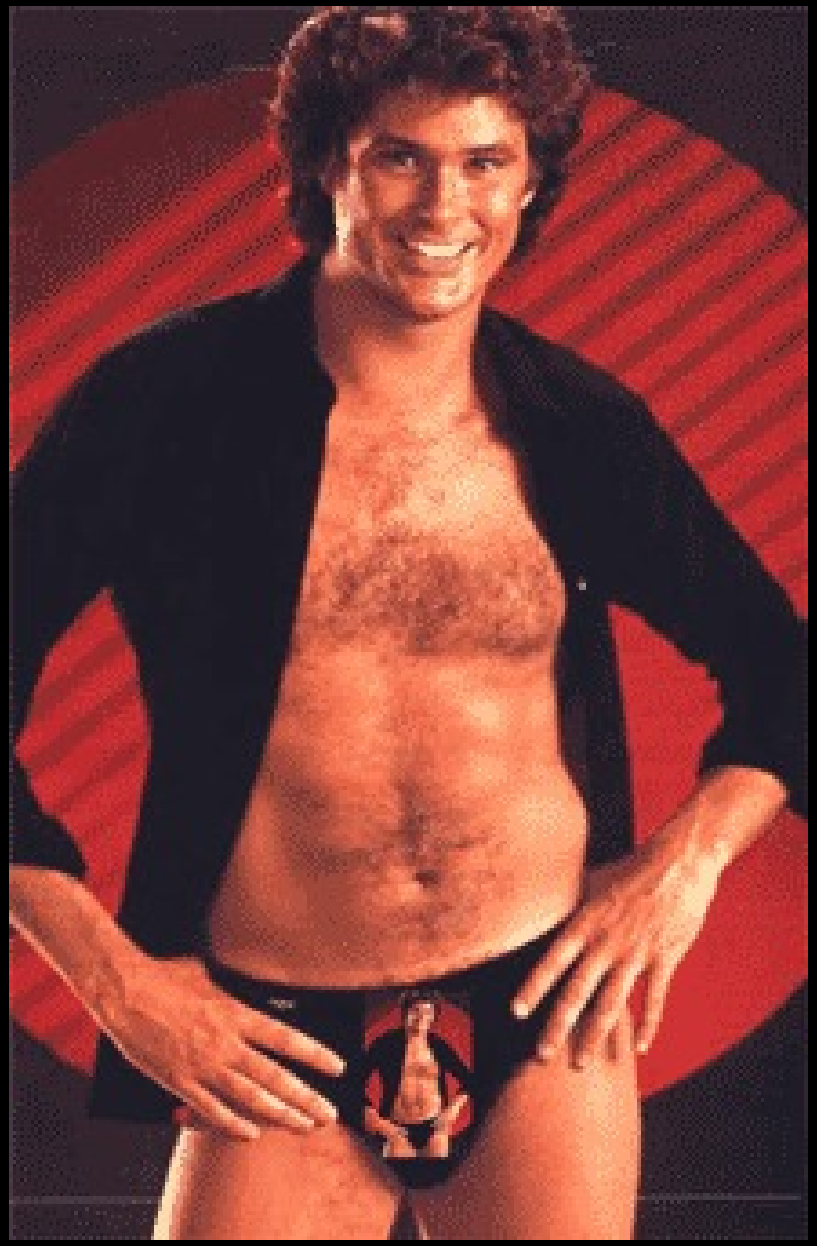












- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

- induction
- induction mathématique (raisonnement par récurrence)
- fonction récurive et type récursif

Factorielle

- $n! = 1 \times 2 \times 3 \times \cdots \times n$
- $n! = \begin{cases} 1 \text{ si } n=0 \\ n \times (n-1)! \text{ si } n \geqslant 1 \end{cases}$
- `fac(n): si n = 0 alors 1 sinon n * (fac (n - 1))`
- `fac(n): si n = 0 alors 1 sinon (fac (n + 1)) / (n + 1)`

Factorielle

- $n! = 1 \times 2 \times 3 \times \cdots \times n$
- $n! = \begin{cases} 1 \text{ si n=0} \\ n \times (n-1)! \text{ si } n \geqslant 1 \end{cases}$
- `fac(n): si n = 0 alors 1 sinon n * (fac (n - 1))`
- `fac(n): si n = 0 alors 1 sinon (fac (n + 1)) / (n + 1)`

Factorielle

- $n! = 1 \times 2 \times 3 \times \cdots \times n$
- $n! = \begin{cases} 1 \text{ si n=0} \\ n \times (n-1)! \text{ si } n \geqslant 1 \end{cases}$
- `fac(n): si n = 0 alors 1 sinon n * (fac (n - 1))`
- `fac(n): si n = 0 alors 1 sinon (fac (n + 1)) / (n + 1)`

Factorielle

- $n! = 1 \times 2 \times 3 \times \cdots \times n$
- $n! = \begin{cases} 1 \text{ si n=0} \\ n \times (n-1)! \text{ si } n \geqslant 1 \end{cases}$
- `fac(n): si n = 0 alors 1 sinon n * (fac (n - 1))`
- `fac(n): si n = 0 alors 1 sinon (fac (n + 1)) / (n + 1)`

Factorielle

```
fac1 :: Integer -> Integer
fac1    0       =  1
fac1    n       =  n * (fac (n-1))
```

```
*Main> fac1 100
933262154439441526816992388562667004907159682643816214685929638952175

999932299156089414639761565182862536979208272237582511852109168640000

0000000000000000000
```

Factorielle

```
fac1 :: Integer -> Integer
fac1    0       =  1
fac1    n       =  n * (fac (n-1))
```

```
*Main> fac1 100
933262154439441526816992388562667004907159682643816214685929638952175

999932299156089414639761565182862536979208272237582511852109168640000

000000000000000000000
```

Factorielle

```
fac2 n = fac_boucle n 1
  where
    fac_boucle 0 accu = accu
    fac_boucle k accu = fac_boucle (k - 1) (k * accu)
```

Factorielle

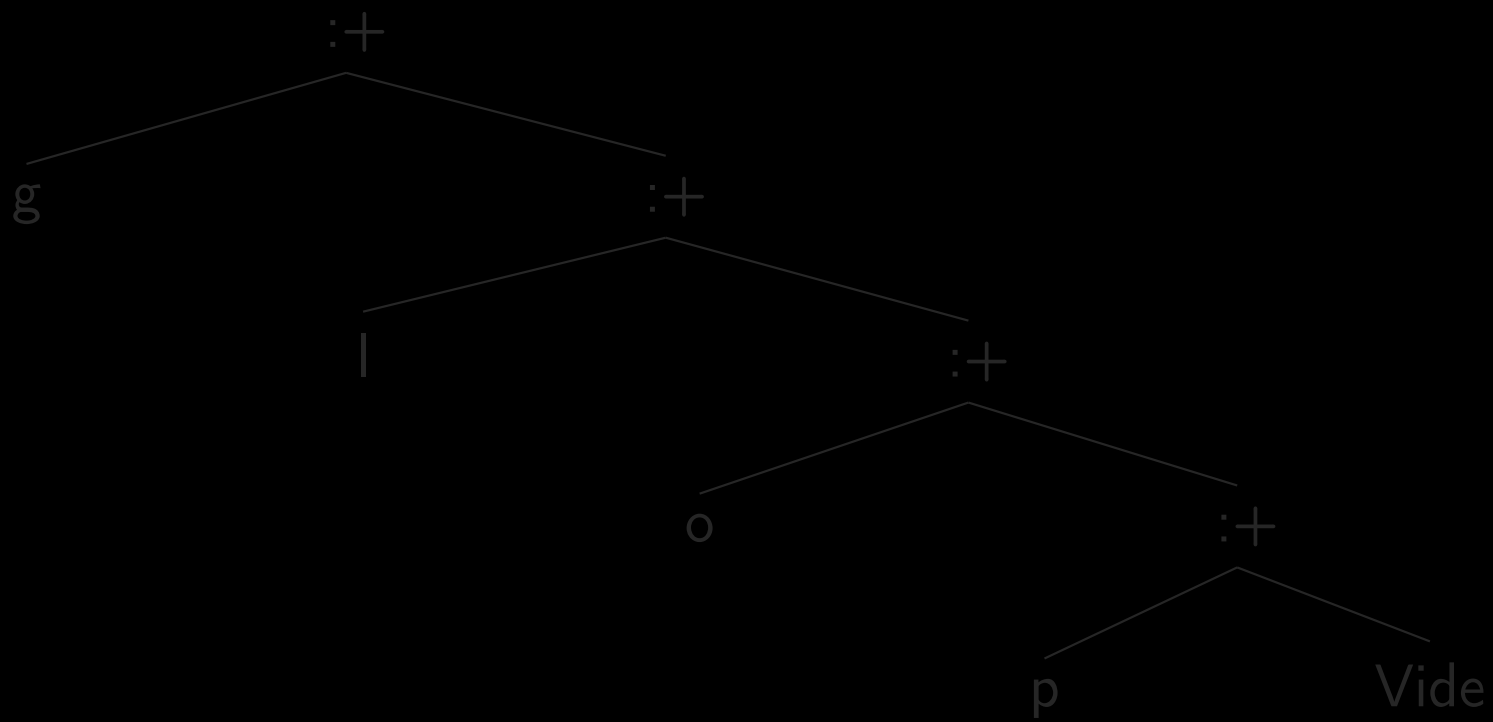
```
fac3 :: Integer -> Integer
fac3 n = foldl (*) 1 [1..n]
```

Factorielle

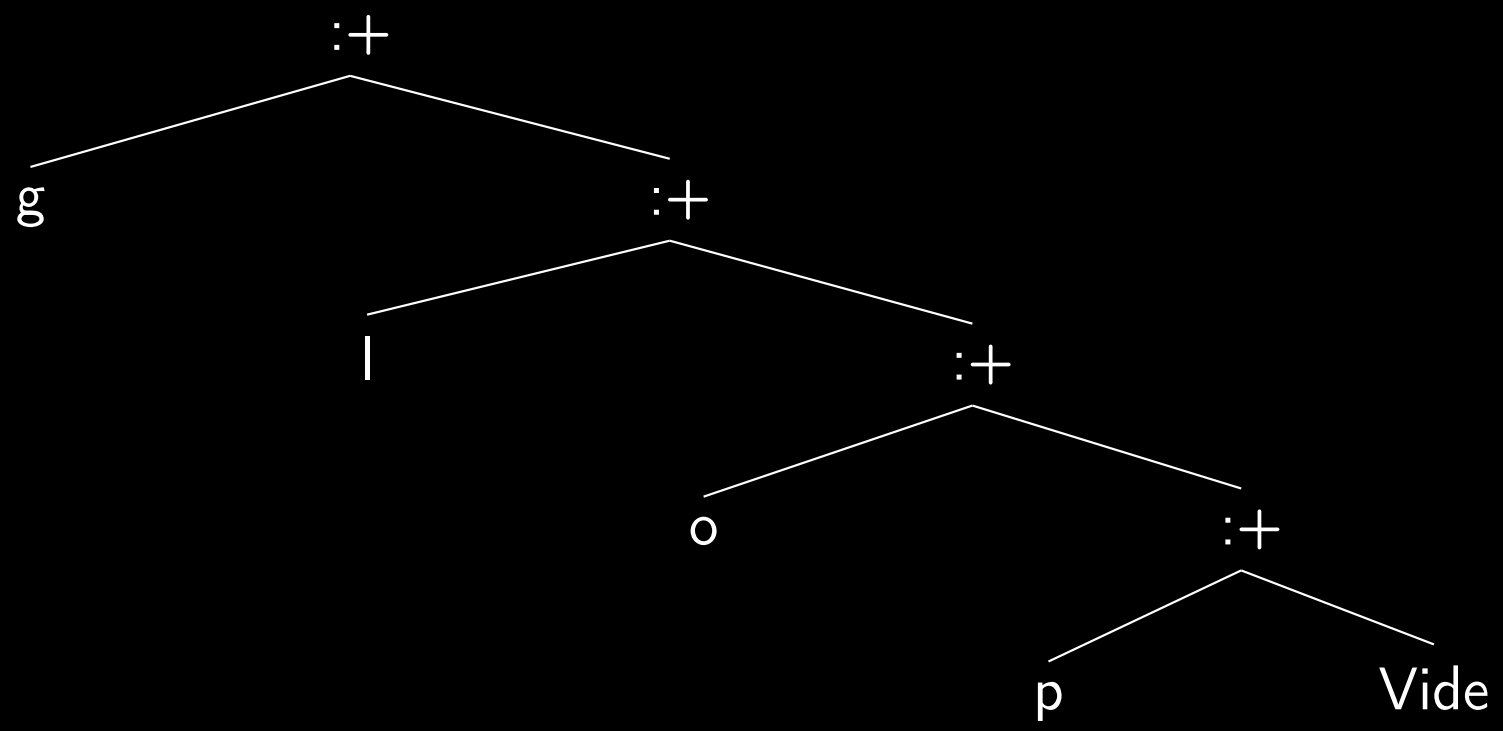
```
fac4 :: Integer -> Integer
fac4 n = product [1..n]
```

Qu'est-ce qu'un mot ?

« glop »



« glop »



```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
```

```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
  deriving (Show)
```

```haskell
infixr :+

data Mot =
  Vide
  | Char :+ Mot
  deriving (Show, Eq)
```

```
*Main> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
*Main> m
'g' :+ ('l' :+ ('o' :+ ('p' :+ Vide)))
```

```
*Main> :t m
m :: Mot
```

```
*Main> let m = 'g' :+ 'l'  :+ 'o' :+ 'p' :+ Vide
*Main> m
'g' :+ ('l' :+ ('o' :+ ('p' :+ Vide)))
```

```
*Main> :t m
m :: Mot
```

Sélecteurs

```
tete :: Mot   -> Char
-- renvoie le premier caractère d'un mot avec un filtrage par motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

```
*Main> tete m
'g'
```

Et la queue ?

Sélecteurs

```
tete :: Mot   -> Char
-- renvoie le premier caractère d'un mot avec un filtrage par motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

```
*Main> tete m
'g'
```

Et la queue ?

Sélecteurs

```
tete :: Mot   -> Char
-- renvoie le premier caractère d'un mot avec un filtrage par motif
tete    Vide  =  error "Mot vide !"
tete (t :+ q) =  t
```

```
*Main> tete m
'g'
```

Et la queue ?

Testeurs

```
estVide :: Mot -> Bool
estVide m = m == Vide
```

$$\neg(\text{estVide } m) \leftrightarrow (m = (\text{tete } m) :+ (\text{queue } m))$$

Testeurs

```
estVide :: Mot -> Bool
estVide m = m == Vide
```

$$\neg(\text{estVide } m) \leftrightarrow (m = (\text{tete } m) :+ (\text{queue } m))$$

Construire une fonction définie sur un type récursif

On voudrait compter le nombre de « a » dans un mot.

```
nba :: Mot -> Int
-- calcule le nombre de 'a' dans un mot
```

Construire une fonction définie sur un type récursif

On voudrait compter le nombre de « a » dans un mot.

```
nba :: Mot -> Int
-- calcule le nombre de 'a' dans un mot
```

- Si le mot est vide, alors son nombre de 'a' est 0.
- Sinon, le mot est construit par `t :+ q`.
  Si `t` = 'a', alors `nba mot` = 1 + `nba q`, sinon, `nba mot` = `nba q`.

```
nba Vide      = 0
nba (t :+ q) = (nba q) + (if t == 'a' then 1 else 0)
```

- Si le mot est vide, alors son nombre de 'a' est 0.
- Sinon, le mot est construit par `t :+ q`.
  Si `t` = 'a', alors `nba mot` = 1 + `nba q`, sinon, `nba mot` = `nba q`.

```
nba Vide      = 0
nba (t :+ q) = (nba q) + (if t == 'a' then 1 else 0)
```

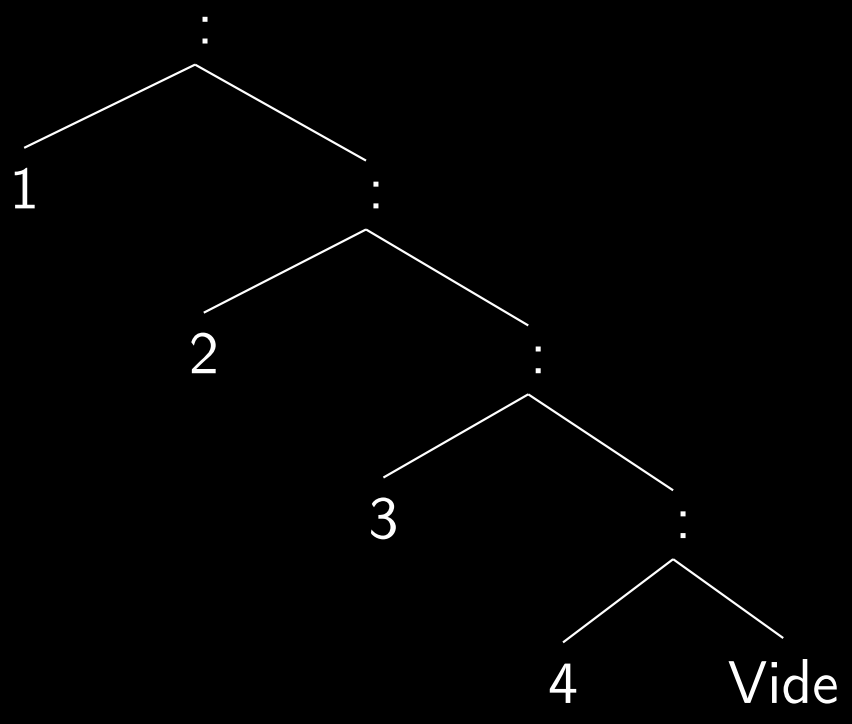
Filtrage par motif

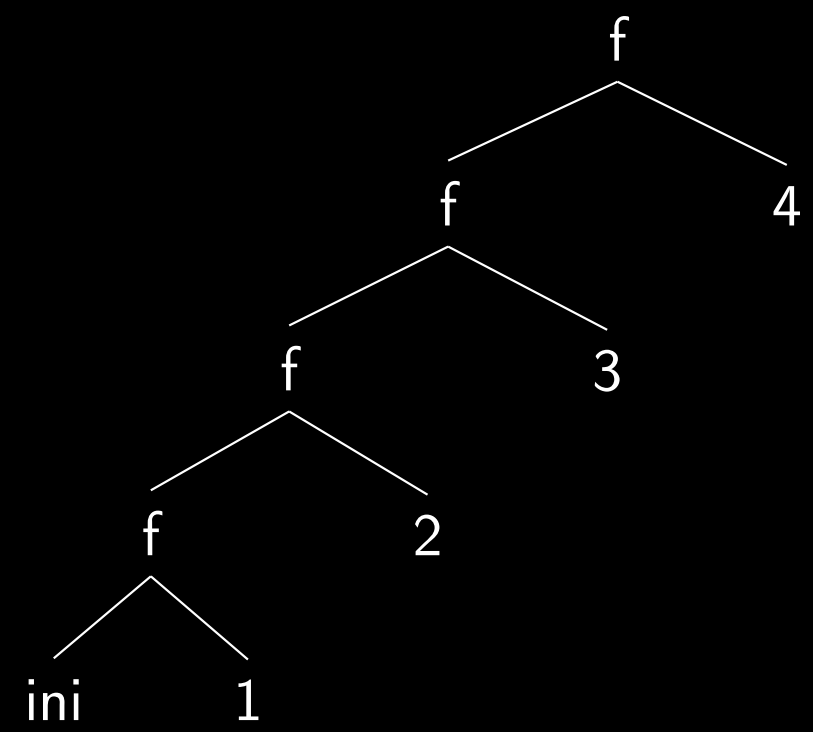
```
nba2 Vide = 0
nba2 (t :+ q)
  | t == 'a' = (nba2 q) + 1
  |otherwise =  nba2 q
```

Case

```
nba3 mot = case mot of Vide      -> 0
                       (t :+ q) -> (nba3 q) + (if t == 'a' then 1
                                               else 0)
```

Pliage





Pliage

```
nba4 :: Mot -> Int
-- calcule le nombre de 'a' dans un mot par pliage
nba4 mot =
  pliage (\accu t -> accu + (if t == 'a' then 1 else 0)) 0 mot
```

Pliage

```
pliage :: (a -> Char -> a) -> a -> Mot -> a
-- adapte foldl aux mots
pliage fonc ini Vide     = ini
pliage fonc ini (t :+ q) = pliage fonc (fonc ini t) q
```

Applique

Mettre en majuscule.

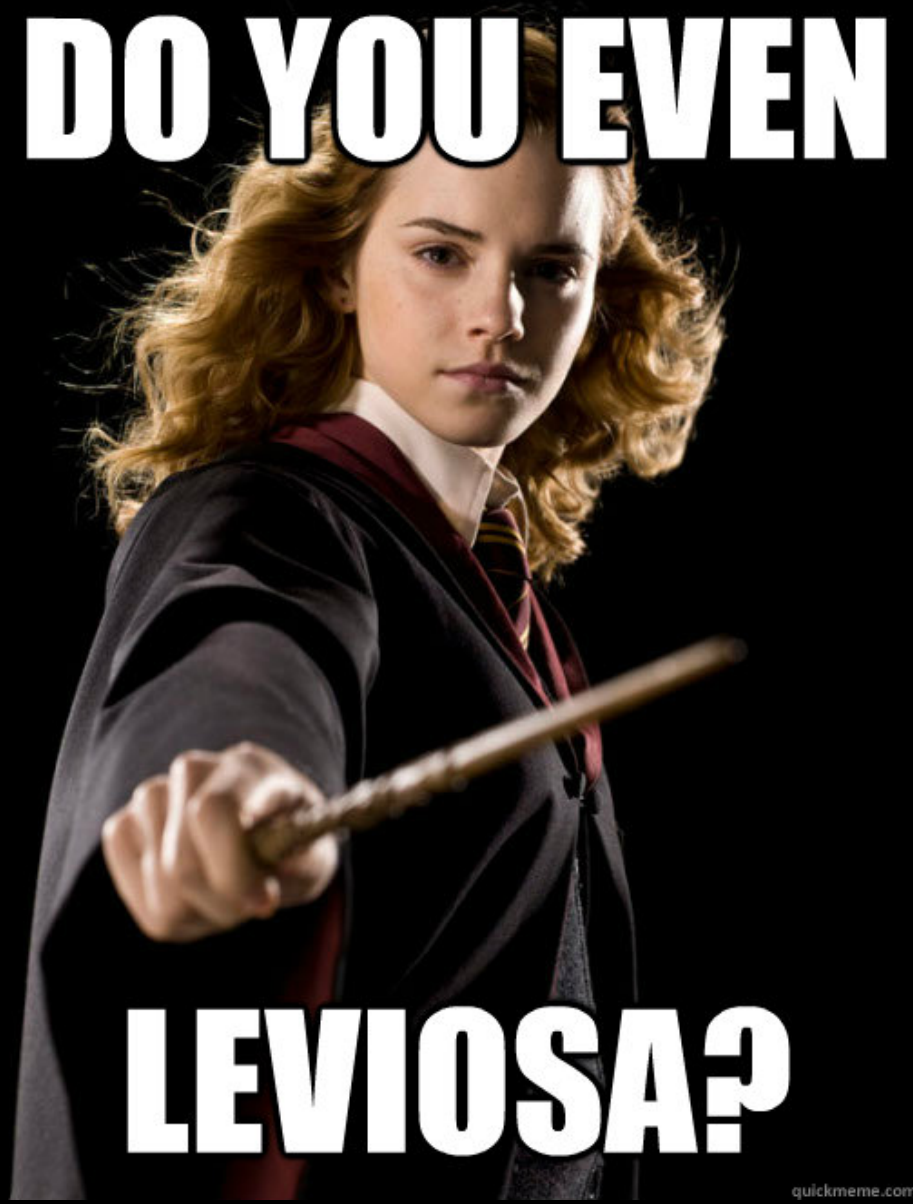
```
*Main> import Data.Char
*Main Data.Char> toUpper 'g'
'G'
```

Applique

Mettre en majuscule.

```
*Main> import Data.Char
*Main Data.Char> toUpper 'g'
'G'
```

glop -> GLOP ?


DO YOU EVEN
LEVIOSA?
quickmeme.com