

64 EXERCICES DE PROGRAMMATION EN HASKELL

L.Gacogne 2001

Le langage haskell

C'est un langage fonctionnel typé, de la famille ML (1977 Université St Andrews en Ecosse), inspiré du λ -calcul. Haskell (1987 en l'honneur de Haskell Curry) est fonctionnel, comme Lisp (1965 Mc Carthy), Scheme et Hope pour des passages par valeur ainsi que Caml (1977 Inria en France), Clean ou Miranda (très analogue à Haskell, 1985 D.Turner), ce qui signifie que l'on déclare pour l'essentiel des fonctions au sens mathématique. Comme Caml, il possède des types de données très rigoureux, et non seulement les contrôle comme dans la plupart des langages, mais les calcule automatiquement, ce qui évite presque toujours au programmeur de les déclarer définissant ses fonctions de la façon abstraite la plus générale possible.

Les points forts de Haskell qui en font un des langages les plus évolués actuellement, sont le caractère purement fonctionnel (y compris les effets de bord), le calcul automatique des types indiquant de plus les contraintes de classe, l'évaluation des paramètres par nécessité (évaluation paresseuse) ce qui permet, entre autre, de donner des définitions mathématiques d'objets infinis, d'autre part, la surcharge de types est mieux prévue pour les opérateurs, on peut donner des définitions par cas avec paramètres structurés un peu comme en Prolog et enfin la syntaxe est nettement moins lourde qu'en Caml, notamment l'analyseur en deux dimensions peut tenir compte de l'indentation.

Haskell est enseigné à Aix la chapelle, Austin, Bonn, Brisbane, Chalmers, Constance, Glasgow, Hull, Melbourne, Nottingham, Oklahoma, Oxford, Oviedo, Utrecht, St Andrews, Yale, York, Wellington...

Voir : www.haskell.org, www.cs.uu.nl/groups/ST/Software/Haskell

et Bird

Hudak P. The Haskell school of expression, Cambridge University Press

Wadler P. Monads for fonctionnal programming, Springer Verlag LNCS 925, 1995

Voir aussi www.cs.mu.oz.au/mercury pour un langage australien "Mercury" à la fois fonctionnel et déclaratif comme Prolog.

Commandes d'utilisation

Les programmes ne sont que des définitions de fonctions et de données écrits dans un éditeur.

Sous UNIX, MacOS ou Windows :

Ouvrir l'interpréteur HASKELL, inclure le programme "nom.hs" par

| | |
|-------------------|---|
| :load nom.hs | charge le fichier spécifié. En chargeant un fichier "essai.hs" le prompt devient <code>essai></code> |
| :load | enlève toutes les définitions chargées à l'exception du "prelude" |
| :also autre.hs | charge un autre fichier |
| :reload | répète la dernière commande de chargement |
| :edit nom.hs | Ouvre le fichier nom.hs |
| :edit | edite le dernier module |
| :module <module> | pose un module for evaluating expressions |
| :type <expr> | affiche le type d'une expression |
| ?: | affiche cette liste de commandes |
| :set <options> | set command line options |
| :set | help on command line options |
| :names [pat] | édite tous les noms contenus |
| :browse <modules> | donne la liste des fonctions et leur type dans le module indiqué |
| :gc | force le garbage collector |
| :quit | exit Hugs interpreter |

Premiers pas avec l'interpréteur d'expressions

Nous verrons des fonctions toutes curryfiées dont l'avantage est d'appliquer toujours une fonction à un seul argument dont l'image est éventuellement une autre fonction. Ainsi $x < y = x < y$ définit bien une fonction `integer -> integer -> bool`, c'est à dire que `inferieur 0` est une fonction `integer -> bool`.

Les expressions avec "variables locales" `let ... in ...` ou `... where ...` s'écrivent comme en Caml, pour les listes, le délimiteurs sont les crochets et la virgule le séparateur, une chaîne de caractères n'est que la liste de ses caractères, d'où :

| | |
|--|---|
| <code>let x = 2 in 4*x</code> 🖱 8 | <code>let [x,y,z] = [1,2,3] in x*y+z</code> 🖱 5 |
| <code>2*x + 3*y where (x, y) = (5, 2)</code> 🖱 16 | <code>2*x + 3*y where x = 5; y = 2</code> 🖱 16 |
| <code>let y = 5; f(x) = (x+y)/y in f(2) + f(3)</code> 🖱 3.0 | <code>f(2) + f(3) where y = 5; f(x) = (x+y)/y</code> 🖱 3.0 |
| Premier élément <code>head [1,2,3]</code> 🖱 1 | <code>head "azerty"</code> 🖱 'a' |
| Queue d'une liste <code>tail [1,2,3]</code> 🖱 [2,3] | |
| Dernier élément <code>last [1,2,3]</code> 🖱 3 | |
| Construction de liste <code>1 : [2,3]</code> 🖱 [1,2,3] | |
| Longueur <code>length [3,7,8,9,4,5,6]</code> 🖱 7 | <code>length "qsdf"</code> 🖱 4 |
| Concaténation <code>[1,2,3] ++ [4,5]</code> 🖱 [1,2,3,4,5] | <code>"azerty" ++ "sdf"</code> 🖱 "azertysdf" |
| Appartenance <code>elem 4 [2,3,4,5]</code> 🖱 True | |
| Énumération <code>[1..10]</code> 🖱 [1,2,3,4,5,6,7,8,9,10] | |
| Tête et queue <code>take 4 [9,8,7,6,5,4,3,2,1]</code> 🖱 [9,8,7,6] | <code>drop 4 [9,8,7,6,5,4,3,2,1]</code> 🖱 [5,4,3,2,1] |
| N-ième élément (à partir du rang 0) | <code>[1,2,3,4,5,6,7,8,9] !! 4</code> 🖱 5 |
| Itérateurs prédéfinis <code>sum [1,3,8]</code> 🖱 12 | <code>sum [1..20]</code> 🖱 210 |
| <code>product [1,8,5]</code> 🖱 40 | <code>product [1..8]</code> 🖱 40320 |
| Opération miroir <code>reverse "azerty"</code> 🖱 "ytreza" | <code>reverse [1,2,3,4,5,6]</code> 🖱 [6,5,4,3,2,1] |
| Coordonnées d'un couple | <code>fst (4,6)</code> 🖱 4 <code>snd (4,6)</code> 🖱 6 |
| Liste des couples de même rang | <code>zip [1,2,3] [4,5]</code> 🖱 [(1,4),(2,5)] |
| Couple de listes à partir de la liste des couples | <code>unzip [(1,2),(3,4)]</code> 🖱 ([1,3],[2,4]) |
| "Zipage" et application d'un opérateur binaire préfixé | |
| <code>zipWith (+) [1,2,3] [4,5,6]</code> 🖱 [5,7,9] | <code>zipWith div [20,25,23] [5,3,7]</code> 🖱 [4,8,3] |
| Exemple de filtrage <code>[x x <- [1..12], even x]</code> 🖱 [2,4,6,8,10,12] | |
| ... n'est évalué que ce qu'il faut ainsi : | <code>take 5 [x x <- [1..1000000], odd x]</code> 🖱 [1,3,5,7,9] |
| Attention, un symbole spécial de génération <code><-</code> est obligatoire, indiquant en quelque sorte sur quoi se font les boucles, (ce pourrait être la propriété <code>elem x ...</code> dans une prochaine version). | |
| Produit cartésien <code>[(x, y) x <- [1,2], y <- [1,2,3]]</code> 🖱 [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)] | |
| Filtre d'emmagasinage tant qu'une propriété est satisfaite | <code>takeWhile odd [3,5,7,2,9,4,1]</code> 🖱 [3,5,7] |
| Filtre de retrait tant qu'une propriété est satisfaite | <code>dropWhile even [4,2,6,3,4,8,13]</code> 🖱 [3,4,8,13] |
| Application d'une fonction unaire aux éléments d'une liste | |
| <code>map length ["a", "la", "mer", "nous", "avons", "trempe", "cruement", "quelques", "gentilles", "allemandes", "stupidement", "bouleversees"]</code> 🖱 [1,2,3,4,5,6,7,8,9,10,11,12] | |
| Exemple d'application d'une fonction anonyme ou λ -abstraction <code>> map (\x -> x+1) [2,3,5]</code> 🖱 [3,4,6] | |
| <code>map</code> peut être redéfini par <code>map _ [] = []</code> et <code>map f (x:q) = f x : map f q</code> . Noter que les priorités sont plus évoluées qu'en Caml car l'infixe <code>comme</code> : est toujours moins prioritaire que les fonctions unaires. | |
| Itération d'un opérateur (avec un élément initial neutre ou pas) par la droite ou la gauche <code>> foldr (+) 0 [1,2,3]</code> 🖱 6 | |
| Application à la concaténation <code>foldr (:) [1,2,3] [4,5]</code> 🖱 [4,5,1,2,3] | |
| ... à l'appâtissement <code>foldr (++) [] [[1,2], [3,4,5], [7,8,9]]</code> 🖱 [1,2,3,4,5,7,8,9] | |
| Exécution de $((30 / 3) / 2) / 5$ par <code>foldl (/) 30 [3, 2, 5]</code> 🖱 1.0 | |
| Exécution de $15 / (20 / (10 / 2))$ par <code>foldr (/) 2 [15, 20, 10]</code> 🖱 3.75 | |
| On pourrait redéfinir <code>som = foldl (+) 0</code> et <code>conc = foldl (++) []</code> | |
| Alors <code>som [1,2,3,4,5,6]</code> 🖱 21 <code>conc [[1,2],[5,6,7],[9,6,3]]</code> 🖱 [1,2,5,6,7,9,6,3] | |
| Boucle : <code>until p f x = if p x then x else until p f (f x)</code> est prédéfini, exemple : | |
| <code>until (\x -> (x==5)) succ 0</code> 🖱 5 | |

Typage, la commande

`:set +t` permet de voir le type des expressions retournées (`:set -t` pour le retour au mode normal) :: signifie l'appartenance à un type.

Exemple de triplet hétérogène `(1, True, "ratp")` 🖱 (1,True,"ratp") :: (Integer,Bool,[Char])

Racine carrée d'un Integer ou d'un Double `sqrt 25` 🖱 5.0 :: Double

Arithmétique et opérateurs préfixes et infixes

Un opérateur préfixe tel que `modulo` peut être utilisé de façon infixe, s'il est entre apostrophes

`mod 23 7` 2 `23 `mod` 7` 2 `div 23 7` 3 `23 `div` 7` 3
 A contrario, un opérateur défini de façon infixé comme + peut être utilisé en préfixe avec des parenthèses
`(++) [1,2,3] [4,5,6]` [1,2,3,4,5,6] `4 + 6` 10 `(+) 4 6` 10
`min 7 2` 2 :: Integer `max 4 5.2` 5.2 :: Double `7 `min` 5` 5 :: Integer
`numerator (52/10)` 26 `denominator (3/9)` 3

Les opérateurs numériques disponibles sont en grand nombre tels que min, max, round, floor, ceiling, sin, cos, exp, log
 Composition de fonction grâce au point (mais maintenir des parenthèses) par exemple :

`demi x = x / 2`
`carre x = x*x`
`(demi.carre) 7` 24.5 `(carre.demi) 7` 12.25 `(log.exp) 5` 5.0

Les fonctions any = or.map et all = and.map sont prédéfinies, exemple :

`all odd [1,3,5,7]` True `any odd [2,4,6]` False

Définitions de fonctions

La forme générale d'une définition est <nom de fonction> <paramètres> = <corps de la définition>, mais elle peut être définie par cas comme la fonction "signe" ci-dessous (attention, sans le =), ou même par plusieurs équations où les paramètres sont structurés de façon à filtrer les différents cas. A ce propos, le joker _ désigne comme en Caml et Prolog un objet quelconque, on pourrait redéfinir head (x : _) = x et tail (_ : q) = q. L'analyseur Haskell est "à deux dimensions", ce qui signifie qu'en indentant correctement les cas, le séparateur ; et les délimiteurs { } deviennent facultatifs.

signe x | x > 0 = 1
 | x < 0 = -1
 | x == 0 = 0

Définition directe de la factorielle : `fac n = if n == 0 then 1 else n * fac (n-1)`
 Avec deux équations : `fac 0 = 1`

En définissant une fonction par cas : `fac n = n * fac (n-1)`
 `fac = \n -> case n of`
 `0 -> 1`
 `(m+1) -> (m+1) * fac m`

Ou encore, plus classiquement : `fac n | n==0 = 1`
 | otherwise = n * fac (n-1)

Dans tous les cas `fact 8` 40320

Exemple de deux définitions mutuellement récursives :

`pair n = if n == 0 then True else impair (n-1) ; impair n = if n == 0 then False else pair (n-1)`

Exemple de résolution par dichotomie, utilisation de "variables locales"

Le principe de la dichotomie, dans le but de résoudre f(x) = 0 sur un intervalle [a, b] où on sait que f s'annule une seule fois, est de diviser en deux cet intervalle au moyen du milieu c, et de reconnaître dans laquelle des deux moitiés, f va s'annuler.

`dicho f a b eps = let c = (a + b) / 2 in if abs (b - a) < eps then c`
 `else if (f a) * (f c) < 0 then dicho f a c eps else dicho f c b eps`

Application au nombre d'or > `dicho (\ x -> (x*x - x - 1)) 0 2 0.001` 1.61767578

Triplets Pythagoriciens Ce sont les triplets d'entiers non nuls tels que $x^2 + y^2 = z^2$.

`triads n = [(x,y,z) | let ns = [1..n], x<-ns, y<-ns, z<-ns, x*x+y*y == z*z]`
`triads 30` [(3,4,5), (4,3,5), (5,12,13), (6,8,10), (7,24,25), (8,6,10), (8,15,17), (9,12,15), (10,24,26), (12,5,13), (12,9,15), (12,16,20), (15,8,17), (15,20,25), (16,12,20), (18,24,30), (20,15,25), (20,21,29), (21,20,29), (24,7,25), (24,10,26), (24,18,30)]

Différence (retrait d'une seule occurrence) `dif l [] = 1`
 `dif l (x:m) = if elem x l then dif (ret x l) m else dif l m`

`dif [4,2,3,5,6,2] [4,5,2,8]` [3,6,2]

Retrait de la première occurrence de x dans une liste `ret x [] = []`
 `ret x (y : q) = if x == y then q else y : (ret x q)`

Aplatissement d'une liste `aplatur [] = []`
 `aplatur ([]:q) = aplatur q`
 `aplatur ((a:q):m) = a:aplatur (q:m)`

`aplatur [[1, 2], [3, 4, 5], [6, 7], [8]]` [1,2,3,4,5,6,7,8]

Tri par segmentation Consiste à choisir un élément pivot (on prend le premier p de la liste) et à séparer les éléments qui lui sont plus petits et ceux qui lui sont plus grands. Ces deux listes triées, il ne reste plus qu'à concaténer le tout (Complexité en n.ln n en moyenne).

`triseq [] = []`

```

triseq (p:q) = triseq [x | x <- q, x <= p] ++ [p] ++ triseq [x | x <- q, x > p]
triseq [5,3,2,6,8,7,4,1,2,5,6,9,0,9,8] [0,1,2,2,3,4,5,5,6,6,7,8,8,9,9]

```

Tri par insertion Le tri par insertion consiste à trier (suivant le même algorithme) la liste privée de son premier élément, puis à y insérer cet élément qui avait été mis de côté (Complexité quadratique).

```

ins x [] = [x]
ins x (y:q) = if x < y then x:y:q else y:(ins x q)
triins [] = []
triins (x:q) = ins x (triins q)
ins 7 [2,5,8,9] [2,5,7,8,9]
triins [5,9,8,0,3,6,4,7,1,5,2,9] [0,1,2,3,4,5,5,6,7,8,9,9]

```

Tri par fusion Le tri par fusion consiste à fusionner (en respectant l'ordre) deux morceaux triés (suivant le même algorithme) de la liste de départ. Les deux morceaux peuvent être pris comme les éléments de rangs pairs ou impairs ce que produit la fonction "separ" (Complexité en n.ln n).

```

fusion l [] = l
fusion [] l = l
fusion (x:p) (y:q) = if x < y then x:(fusion p (y:q)) else y:(fusion (x:p) q)
trifus [] = []
trifus [x] = [x]
trifus l = fusion (trifus g) (trifus d) where (g, d) = separ l [] []
separ [] p i = (p, i)
separ (x:q) p i = separ q (x:i) p
fusion [1,3,5,8,9] [0,1,2,3,4,5,7] [0,1,1,2,3,3,4,5,5,7,8,9]
separ [0,1,2,3,4,5,6] [] [(6,4,2,0),[5,3,1]]
trifus [4,2,6,8,9,1,2,5,8,3] [1,2,2,3,4,5,6,8,8,9]

```

Parties d'un ensemble

Les parties d'une liste sont celles ne contenant pas un élément quelconque a de la liste, et celles contenant cet a, c'est à dire les mêmes auxquelles on rajoute a.

```

parties [] = [[]]
parties (a:q) = let pq = parties q in pq ++ (map (\ x-> a:x) pq)
parties [1,2,3] [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]

```

Permutations

Les permutations d'une liste sont toutes celles obtenues en prenant successivement un élément x de la liste suivi de toutes les permutations p possibles des autres.

```

perm [] = [[]]
perm e = [x : p | x <- e, p <- perm (ret x e) ]
perm [1,2,3] [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

```

Intégrale par Simpson

Formule correspondant à des approximations paraboliques

```

simpson f a b n = (h / 3) * (f(a) - f(b) + 2 * simpson' 0 a)
  where simpson' s x = if x >= b then s else simpson' (s + 2 * f (x + h) + f (x + 2*h)) (x + 2*h)
        h = (b - a)/(2*n)

```

```

simpson (\ x -> 4/(1+x*x)) 0 1 20 3.14159265

```

Currification

Les fonctions ne sont définies qu'à une seule variable. Par exemple (sauf si on donne explicitement l'unique variable comme un couple), une fonction de deux arguments n'est jamais qu'une fonction à un argument (le premier) dont le résultat est lui-même une fonction à un argument (le second). Ainsi si f est (E*F -> G) alors (Curry f) sera (E -> (F -> G)) et en Haskell comme en Caml, l'écriture f x y z inférra que f est de type (E -> (F -> (G -> H))) et cette expression sera prise pour ((f x) y) z avec (f x) de type F -> (G -> h) et ((f x) y) de type (G -> H). On adopte la convention du parenthésage à gauche, c'est à dire que f g x signifie (f g) x et est interprété. Ainsi une fonction (a -> b) -> (c -> d) peut s'écrire (a -> b) -> c -> d. Une fonction currifiée reçoit donc ses arguments un à un, mais aussi elles sont applicables partiellement et plus rapides (des constructions de n-uplets en moins) et privilégiées par la compilation. A première vue la currification est une opération ramenant toutes les fonctions à des compositions de fonctions unaires, si f a été définie comme une fonction à deux arguments. Curry et uncurry sont des fonctions prédéfinies, c'est :

```

curry f = \ x -> \ y -> f(x, y)      dont le type est (a * b -> c) -> a -> b -> c
uncurry f = \ (x, y) -> f(x) y      de type (a -> b -> c) -> a * b -> c

```

Evaluation paresseuse et objets infinis

Pour retourner, une valeur à une expression demandée, il faut "réduire" cette expression en suivant un certain ordre. La réduction vers l'expression irréductible (ou normale) n'est pas toujours possible, et si elle existe n'est pas forcément atteinte. En profondeur d'abord (innermost) signifie que pour chaque $f(x)$, l'argument x doit être évalué avant la fonction f sur son argument x . C'est, avec la lecture de gauche à droite, la méthode généralement utilisée du passage de paramètre par valeur (ou par adresse, auquel cas il faut aller chercher cette valeur dans un endroit précis). C'est d'ailleurs cet ordre "naturel" qui légitime les notations polonaises ou postfixées qui permettent, comme en Forth, de commencer l'évaluation d'une expression au fur et à mesure de sa lecture. Par contre une évaluation "par l'extérieur" consiste à tenter l'évaluation de f d'abord (outermost), et à ne faire celle de x que si c'est nécessaire. Dans un cas tel que $\text{fac}(100) - \text{fac}(100)$, il est évident que si on peut savoir que l'opérateur - renvoie 0 pour les mêmes expressions, on n'a pas besoin de les calculer, mais on n'en est pas là. L'évaluation paresseuse est une évaluation par l'extérieur et de gauche à droite avec partage de l'expression. Partage signifie qu'une expression évaluée est temporairement conservée pour les éventuelles autres instances de la même expression.

Exemple, en définissant $f(x) = f(x+1)$, $g(x) = 3$, $r = g(f\ 4)$, un langage ordinaire ne pourrait donner la valeur 3 pour r . Par ailleurs, dans la séquence suivante $f\ a$ est calculé correctement alors que la demande de a provoque une erreur.

```
a = 1/0
```

```
f x = 1
```

```
f a  1
```

```
a  Program error: {primDivDouble 1.0 0.0}
```

L'évaluation paresseuse est possible avec `df` ou `defmacro` en Lisp, et est en fait utilisée dans beaucoup de langages pour certaines fonctions telles que `if`. En effet si on définit `iff x y z = if x then y else z` et `fac n = iff (n == 0) 1 (n * fac (n-1))` tout va bien, mais dans un autre langage, l'application de "iff" suppose ses trois arguments préalablement évalués d'où un enchaînement infini de calculs.

L'inconvénient est néanmoins patent dans certaines expressions, ainsi `sqr(3+4)=sqr 7 = 7*7 = 49` est meilleur en passage par valeur que `sqr(3+4) = (3+4)*(3+4) = 7*(3+4) = 7*7 = 49` en passage paresseux.

Cependant le gros avantage de l'évaluation par nécessité, outre l'économie d'évaluations, est de pouvoir donner des définitions correctes d'objets infinis, quitte à n'en tirer évidemment que des portions finies. Ainsi la liste `r = 0 : r` est parfaitement définissable et on peut voir son début, ou son 53-ième élément :

```
take 7 r  [0,0,0,0,0,0,0]          r !! 53  0
```

Nombres premiers

On définit la liste infinie des entiers à partir de n , puis le crible d'une liste d'entiers comme la liste conservant le premier de la liste suivi des autres d'où on a retiré les multiples du premier.

```
depuis n = n : depuis (n+1)
```

```
retirmultiples d (n:q) = if mod n d == 0 then lasuite else n:lasuite
                        where lasuite = retirmultiples d q
```

```
crible (x:q) = x:crible (retirmultiples x q)
```

```
nbpremiers = crible (depuis 2)
```

```
take 20 nbpremiers  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

```
premier n = elem n nbpremiers
```

Attention cette définition est possible mais ne répondra jamais False, il faut jouer sur l'ordre en définissant plutôt :

```
premier n = elem n (jusqua n nbpremiers) where jusqua n (p : q) = if p > n then [] else p : (jusqua n q)
```

```
premier 53  True          premier 54  False
```

Grâce à la fonctionnelle "iterate" qui produit la liste de toutes les images $f^{(n)}(x)$ pour n entier, on a une définition encore plus courte des nombres premiers.

```
take 5 (iterate sqrt 65536)  [65536.0, 256.0, 16.0, 4.0, 2.0]
```

```
take 10 (iterate (* 2) 1)  [1,2,4,8,16,32,64,128,256,512]
```

```
puissances n = iterate (* n) 1
```

```
puis n k = (puissances n) !! k
```

```
puis 2 10  1024
```

```
take 7 (iterate reverse [1,2,3])  [[1,2,3],[3,2,1],[1,2,3],[3,2,1],[1,2,3],[3,2,1],[1,2,3]]
```

```
nbpremiers = map head (iterate crible [2..])
```

```
crible (p : q) = [ x | x <= q, mod x p /= 0 ]
```

```
take 20 nbpremiers  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

Liste des diviseurs propres `facteurs n = [i | i <- [1..n-1], mod n i == 0]`

```
facteurs 30  [1,2,3,5,6,10,15]
```

Nombre parfait (égal à la somme de ses diviseurs propres)

```
parfait n = sum (facteurs n) == n
```

```
nbparfaits = filter parfait [(1::Int)..]
```

```
take 3 nbparfaits  [6,28,496]
```

Retour aux factorielles `suitefac = 1 : zipWith (*) suitefac [1..]`

```
take 8 suitefac  [1,1,2,6,24,120,720,5040]
```

```

Suite de Fibonacci      suitefib = 1 : 1 : [u + v | (u, v) <- zip suitefib (tail suitefib)]
                        take 7 suitefib [1,1,2,3,5,8,13]
                        fib n = suitefib !! n
                        fib 7 [1,2,3,5,8,13,21]
Ou encore :            fib n = fibs !! n where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

```

Encore un très bel exemple avec le triangle de Pascal

A partir de la ligne [1], on itère une fonction qui à toute ligne "lgn", associe l'addition terme à terme de cette ligne avec elle-même décalée (précédée de 0) :

```

pascal = iterate (\lgn -> zipWith (+) ([0] ++ lgn) (lgn ++ [0])) [1]
take 5 pascal [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

```

Récurivité terminale

La récursivité terminale est établie dans le calcul d'une fonction f si et seulement si dans tout appel de f avec certaines valeurs, le résultat final sera celui d'un autre appel (avec d'autres valeurs). Cela signifie que le second appel ayant achevé son travail, le premier n'a pas à reprendre la main pour un travail quelconque supplémentaire comme par exemple la composition avec une autre fonction, il a "terminé". L'intérêt de la récursivité terminale pour les langages qui la détectent, c'est que chaque sous-programme (un appel de fonction) n'a pas à conserver son contexte lorsqu'il est déroté sur l'appel suivant puisqu'il a fini et que les valeurs des arguments qu'on lui a passé n'ont plus d'utilité. L'occupation de la mémoire est alors constante comme dans le cas d'un programme itératif.

Une programmation récursive terminale est visible à l'oeil nu dans $f(x) = \text{si } \langle \text{condition} \rangle \text{ alors } \langle \text{valeur} \rangle \text{ sinon } f(\dots)$, elle est récursive non terminale dans $f(x) = \text{si } \dots \text{ alors } \dots \text{ sinon } g(f(\dots))$ puisque le second appel doit être composé avec un traitement supplémentaire g .

Dans la pratique, on prend tous les paramètres entrant en ligne de compte en cours de résolution du problème, comme fac' définie pour r (résultat partiellement construit) et n (entier courant descendant de la donnée initiale jusqu'à 1).

```

fac = fac' 1 where fac' r = \ n -> if n == 0 then r else fac' (n*r) (n-1)
fac 49 [60828186403426756087225216332129537688755283137921024000000000]

```

Point fixe et fonction de Mac Carthy

Si F est l'ensemble des fonctions partiellement définies sur D à valeurs dans D et si f est une fonction croissante (pour l'inclusion des graphes) de F dans F , le théorème de Knaster-Tarski indique que f admet un plus petit point fixe f c'est à dire tel que $f(f) = f$. Si de plus f est continue (vérifiant $f(\bigcup_{k \in \mathbb{N}} f^k(\emptyset)) = \bigcup_{k \in \mathbb{N}} f^k(\emptyset)$ lorsque f_n est une suite croissante au sens de l'inclusion des graphes, alors ce point fixe est $\bigcup_{k \in \mathbb{N}} f^k(\emptyset)$.

Par exemple "fac" est le plus petit point fixe de $f(f)(x) = \text{si } x = 0 \text{ alors } 1 \text{ sinon } x.f(x-1)$ et la fonction de McCarthy ci-dessous est celui de $f(f)(x) = \text{si } x > 100 \text{ alors } x-10 \text{ sinon } f^2(x+11)$, on a alors une suite de fonction partielles :

$f(\emptyset)(x) = \text{si } x > 100 \text{ alors } x - 10$, d'où aussi $f^2(\emptyset)(x) = \text{si } x > 100 \text{ alors } x - 10 \text{ sinon si } x > 99 \text{ alors } x - 9$, et :
 $f^k(\emptyset)(x) = \text{si } x > 100 \text{ alors } x - 10 \text{ sinon si } x > 101 - k \text{ alors } 91$ pour $k \leq 11$ et cette suite devient stationnaire à $f^{20}(\emptyset)(x) = \text{si } x > 100 \text{ alors } x - 10 \text{ sinon } 91$. La fonction f_{91} de Mac Carthy est définie récursivement par :

```

m x = if x > 100 then x-10 else m (m (x+11))

```

La fonction généralisée de McCarthy est $\text{mc}(x, y) = \text{si } x > 100 \text{ alors si } y = 1 \text{ alors } x - 10 \text{ sinon } \text{mc}(x-10, y-1)$
sinon $\text{mc}(x+11, y+1)$

dans laquelle y compte les appels et $f(x, y) = f_{91}(x)$

```

mc x y = if x > 100 then if y == 1 then x-10 else mc (x-10) (y-1) else mc (x+11) (y+1)

```

```

mc 100 2 [91]      mc 123 5 [91]      mc 45 23 [91]

```

La fonction d'Ackerman

C'est l'exemple déjà vu d'une fonction récursive non primitive (mais pas à cause de son double appel car par exemple $f(n, 0) = 10^n$ et $f(n, m+1) = f(f(n, m), m)$ a un double appel et est primitive récursive).

```

ak x y = if x == 0 then y+1 else ak (x-1) (if y == 0 then 1 else ak x (y-1))

```

```

ak 2 3 [9]      ak 2 5 [13]      ak 3 6 [509] -- (commence à devenir long)

```

Algorithme à retour en arrière, parcours d'arborescence

Le problème général du backtracking, est d'arriver à un état satisfaisant une fonction "but", à partir d'un état "init", par le moyen d'une fonction "fils" donnant pour chaque état la liste des états suivants possibles. La fonction bak renvoie ici le couple formé du booléen possible ou non, et de la liste complète des états permettant de passer de "init" à la première solution rencontrée. Ce chemin ch est toutefois à l'envers pour des raisons de commodité d'ajout ou de retrait.

```

bak fils but init = bak' [init] [] where bak' [] ch = (False, ch)
                                   bak' (x : q) ch = if but x then (True, x : ch)
                                                         else if elem x ch then bak' q ch
                                                         else bak' ((fils x) ++ q) (x : ch)

```

Application "au compte est bon"

Il s'agit de réaliser une somme s avec les valeurs contenues dans une liste ln , on va représenter chaque état intermédiaire du problème par le couple formé par la liste lc des nombres choisis et celle lr de ceux restant disponibles, la difficulté est d'écrire le but à atteindre, surtout la liste des "fils" ou états suivants d'un état, et enfin une fonction de visualisation "vue".

```
compte s ln = vue (bak (\ (lc, lr) -> map (\x -> (x : lc, ret x lr)) lr) (\ (lc, _) -> sum lc == s) ([], ln)
  where vue (b, le) = if b then fst (head le) else []
```

En reprenant les définitions plus haut "ret" et "dif",

```
dif l [] = 1
```

```
dif l (x:m) = if elem x l then dif (ret x l) m else dif l m
```

```
ret x [] = []
```

```
ret x (y : q) = if x == y then q else y : (ret x q)
```

```
compte 7 [3,2,1,4,3]  [3,1,3]           compte 7 [3,2,1]  []
```

Application aux tours de Hanoi, ici on représente un état du problème par le triplet des listes de plateaux. Ceux-ci sont numérotés par diamètre et on vérifie par "correct" qu'au dessus de la pile est toujours placé un plateau plus petit que celui du dessous. Fils donne la liste des états possibles.

```
correct (a, b, c) = correct' a && correct' b && correct' c
```

```
  where correct' [] = True
```

```
        correct' [_] = True
```

```
        correct' (x:y:q) = (x < y)           -- il n'est pas nécessaire de vérifier en dessous
```

```
fils (x, y, z) = fils' (x, y, z) ++ (map (\(a, b, c) -> (b, a, c)) (fils' (y, x, z)))
```

```
        ++ (map (\(a, b, c) -> (c, b, a)) (fils' (z, y, x)))
```

```
  where fils' ([], _, _) = []
```

```
        fils' (x:q, b, c) = filter correct [(q, x:b, c), (q, b, x:c)]
```

```
suite n = take n (apres 0) where apres n = (n+1) : (apres (n+1))
```

```
hanoi n = bak fils (\ x -> x == ([], [], suite n)) (suite n, [], [])
```

```
hanoi 3 (On a ici des mouvements inutiles)  (True, [([],[],[1,2,3]), ([1],[],[2,3]), ([], [1],[2,3]), ([2],[1],[3]), ([1,2],[],[3]), ([1,2],[3],[]), ([2],[1,3],[]), ([2],[3],[1]), ([],[2,3],[1]), ([],[1,2,3],[]), ([1],[2,3],[]), ([1],[3],[2]), ([],[1,3],[2]), ([],[3],[1,2]), ([3],[],[1,2]), ([1,3],[],[2]), ([3],[1],[2]), ([2,3],[1],[]), ([1,2,3],[],[])])
```

Le backtrack n'est pas indispensable pour ce problème, voir plus loin.

Application aux reines de Gauss

On remplit un tableau $n*n$ par lignes en disposant une reine par ligne à un certain numéro de colonne. La représentation naturelle sera donc la liste des numéros de colonnes et celle ci sera valide si chaque reine nouvellement placée en x n'est pas en prise avec les précédentes. Par exemple si l'état précédent est $e = [5, 3, 1]$, il faut $x \neq 5$, 5 ± 1 puis $x \neq 3$, 3 ± 2 enfin $x \neq 1$, 1 ± 3 d'où le paramètre r de la fonction valide.

```
valide [] _ = True
```

```
valide (x:y:q) r = if x == y || x == y+r || x == y-r then False else valide (x:q) (r+1)
```

```
fils n e = filter (\ f -> valide f 1) (map (\ x -> x:e) (jusqua n))
```

Remarque sur les types, ici, fils est définie curryfiée à deux arguments, qu'à cela ne tienne, fils n est à un argument :

```
  bak :: Eq a => (a -> [a]) -> (a -> Bool) -> a -> (Bool,[a])
```

```
  fils :: Num a => a -> [a] -> [[a]]
```

```
  fils 7 :: Num a => [a] -> [[a]]
```

```
jusqua n = if n == 0 then [] else n:jusqua (n-1)
```

```
jusqua 7  [7,6,5,4,3,2,1]
```

```
reines n = head (snd (bak (fils n) (\ x -> n == length x) []))
```

```
reines 4  [2,4,1,3]
```

```
reines 5  [2,4,1,3,5]
```

```
reines 6  [2,4,6,1,3,5]
```

```
reines 7  [2,4,6,1,3,5,7]
```

```
reines 8  [5,7,2,6,3,1,4,8]
```

Le taquin

Dans un carré $n*n$, une case est laissée vide, on peut y faire glisser l'une des voisines. Après un certain nombre de tels mouvements, il est possible en partant d'un état initial donné d'arriver à la moitié des configurations possibles. On note I et B les deux états fixés : initial et but, par exemple :

I =

| | | |
|---|---|---|
| 5 | 2 | 8 |
| 3 | 4 | 0 |
| 6 | 7 | 1 |

B =

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 0 | 4 |
| 7 | 6 | 5 |

On adopte une représentation sous forme de liste où 0 figure la case vide et où le n -ième élément (à partir de 0) est le couple (i, j) repérant sa position du numéro de ce rang. La fonction "bak" est modifiée de façon à renvoyer un booléen

uniquement, mais en visualisant les mouvements du chemin ch qui est la liste des états retenus au cas où le problème est faisable. Ainsi les états initial et final :

ei = [(2,3), (3,3), (1,2), (2,1), (2,2), (1,1), (3,1), (3,2), (1,3)]

ef = [(2,2), (1,1), (1,2), (1,3), (2,3), (3,3), (3,2), (3,1), (2,1)]

dist (a, b) (c, d) = abs (a - c) + abs (b - d) -- distance de Hamming

fils e = [swap i e | i <- [1.. length e - 1], valide i e]

where valide i e = (dist (e!!0) (e!!i) == 1)

swap i e = [(e!!i)] ++ (take (i-1) (tail e)) ++ [(e!!0)] ++ (drop (i+1) e)

fils ei  [[(3,3),(2,3),(1,2),(2,1),(2,2),(1,1),(3,1),(3,2),(1,3)], [(2,2),(3,3),(1,2),(2,1),(2,3),(1,1),(3,1),(3,2),(1,3)], [(1,3),(3,3),(1,2),(2,1),(2,2),(1,1),(3,1),(3,2),(2,3)]] -- On vérifie qu'il s'agit des 3 états possibles à partir de ei

Ce problème est très long, il ne mène guère à la solution pour n > 2, on utilise des heuristiques cherchant par exemple à minimiser la distance de Hamming d'avec le but. Pour n = 2, ei = [(2,1),(1,1),(1,2),(2,2)], ef = [(1,1),(1,2),(2,2),(2,1)]

taquin  (True, [[(1,1),(1,2),(2,2),(2,1)], [(2,1),(1,2),(2,2),(1,1)], [(2,2),(1,2),(2,1),(1,1)], [(1,2),(2,2),(2,1),(1,1)], [(1,1),(2,2),(2,1),(1,2)], [(2,1),(2,2),(1,1),(1,2)], [(2,2),(2,1),(1,1),(1,2)], [(1,2),(2,1),(1,1),(2,2)], [(1,1),(2,1),(1,2),(2,2)], [(2,1),(1,1),(1,2),(2,2)]]] -- sont les 10 états dans l'ordre de efinal à einitial.

Le problème des missionnaires

Sur la rive gauche d'un fleuve se trouvent n missionnaires et n cannibales disposant d'une barque contenant k passagers, le but est de faire passer tout le monde sur l'autre rive en respectant la contrainte nb-missionnaires ≥ nb-cannibales en chaque endroit y compris dans la barque. Un état du problème sera le triplet représentatif de la rive gauche (nm, nc, barque), l'état initial étant (n, n, True) et final (0, 0, False)

valide m c = (m == 0) || (c <= m)

interval a b = if a > b then [] else a:(interval (a+1) b)

fils (m, c, True) n k = [(m', c', False) | m' <- interval 0 m, c' <- interval 0 c, m-m'+c-c' <= k, 0 < m-m'+c-c', valide (n-m') (n-c'), valide (m-m') (c-c'), valide m' c']

fils (m, c, False) n k = [(m', c', True) | m' <- interval m n, c' <- interval c n, m'-m+c'-c <= k, 0 < m'-m+c'-c, valide (m'-m) (c'-c), valide (n-m') (n-c'), valide m' c']

mission n k = bak (\ e -> fils e n k) (\ e -> e == (0, 0, False)) (n, n, True)

Exemples (bien sur il faudrait rajouter une visualisation correcte en remettant à l'endroit et de façon explicite les différentes étapes) :

mission 3 2  (True, [(0,0,False), (0,2,True), (0,1,False), (0,3,True), (0,2,False), (2,2,True), (1,1,False), (3,1,True), (3,0,False), (3,2,True), (2,2,False), (3,3,True)])

mission 4 2  (False, ...

mission 4 3  (True, [(0,0,False), (0,2,True), (0,1,False), (0,4,True), (0,3,False), (3,3,True), (2,2,False), (4,3,True), (3,3,False), (4,4,True)])

mission 5 2  (False, ...

mission 5 4  (True,[(0,0,False), (0,2,True), (0,1,False), (0,5,True), (0,4,False), (4,4,True), (3,3,False), (5,5,True)])

Modules et importations

Les modules contiennent diverses déclarations de classes, types et fonctions, ce n'est qu'un ensemble de déclarations. Un module peut parfaitement être sur plusieurs fichiers de même qu'un fichier peut contenir plusieurs modules. Par exemple, on définit le module relatif aux arbres binaires non étiquetés par :

module Arb (Arb(Feuille, Noeud), parcours) where

data Arb a = Feuille a | Noeud (Arb a) (Arb a)

parcours (Feuille x) = [x]

parcours (Noeud g d) = (parcours g) ++ (parcours d)

Le module de base s'appelle "prelude", il contient toutes les fonctions de base, mais bien d'autres sont accessibles par "import", tels que les modules List, Array, Random, IO ...

Classes et types

Les types simples sont nécessaires à une structuration rationnelle et économique des données, ainsi une variable booléenne peut n'être codée que par un bit alors qu'un entier utilise ordinairement 2 octets et un réel (ou double) 4 octets et par exemple une couleur de 0 à 255 sur 1 octet etc.

Les types structurés, listes d'éléments, tableaux, arbres ... correspondent à des façon de ranger en mémoire différent éléments de même type et à y accéder de diverses manières. Par exemple une liste chaînée est représentée par un doublet d'adresse, un tableau par une adresse début, un pas dépendant du type des éléments contenus, et une dimension.

Les classes, elles correspondent cette fois à des regroupements de types d'objets disposant en commun de "méthodes" c'est à dire de fonctions portant le même nom et surtout le même symbole. La classe la plus large prédéfinie est par exemple, Eq avec l'opérateur == d'un type que l'on pourrait noter Eq -> Eq -> Bool, mais en fait noté par Haskell (Eq a) => a -> a -> Bool qu'il faut lire étant donné deux éléments du même type a, un booléen est renvoyé à la condition que le type a fasse partie de la classe Eq. Cette contrainte ou "contexte" sera par exemple aussi indispensable pour la fonction elem :: (Eq a) => a -> [a] -> Bool se lisant "la contrainte Eq a permet que la fonction elem ait un premier argument du type a et un autre du type structuré liste de a". Aux classes, sont attachées des définitions générales de fonctions, lesquelles (avec toujours le même nom) auront des définitions particulières suivant les types "instances" de ces classes.

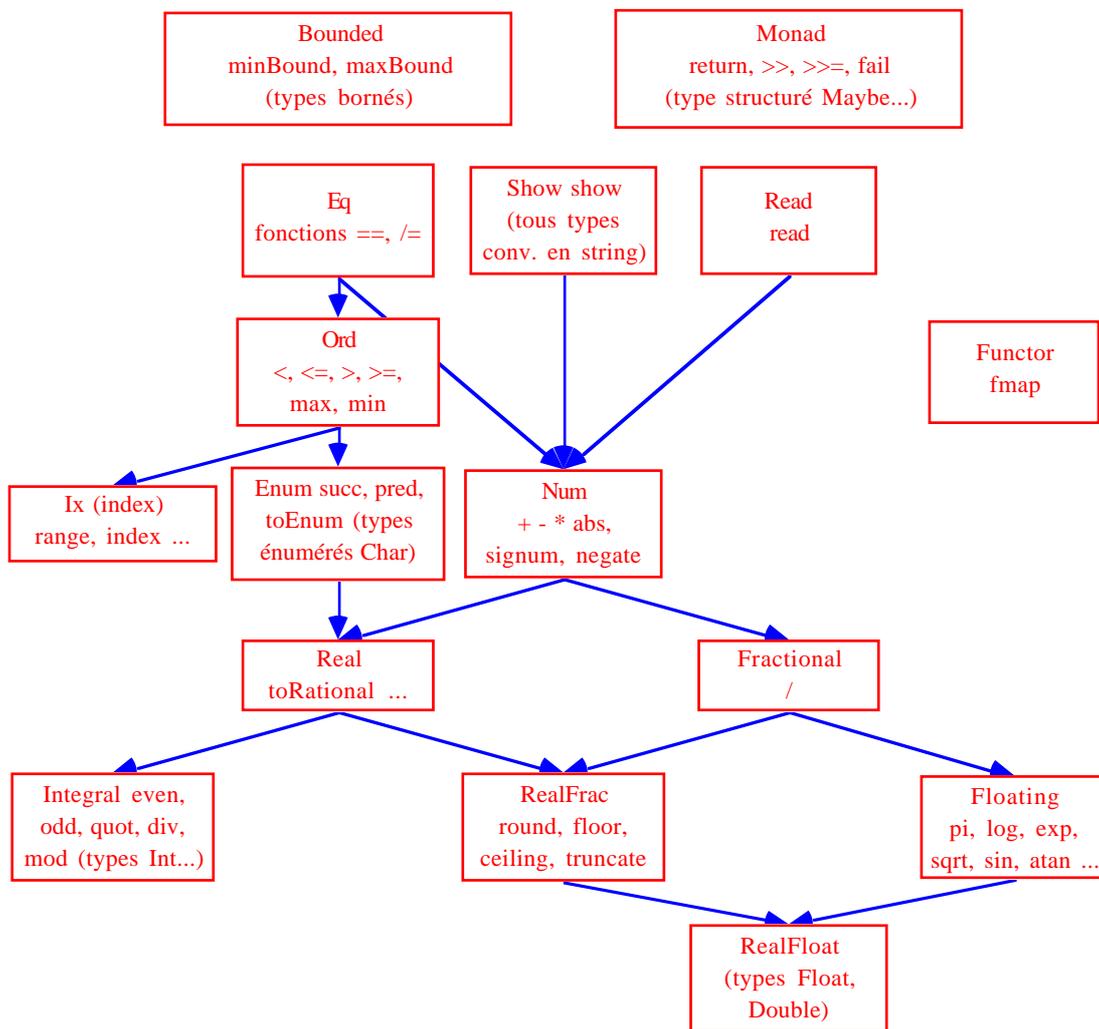
Par exemple sont prédéfinis :

```
Class Eq a where (==), (/=) :: a -> a -> Bool
                x /= y = not (x == y)
```

Puis un héritage pour la "sous-classe" Ord

```
Class (Eq a) => Ord a where (<), (<=), (>), (>=) :: a -> a -> Bool
                          max, min ;; a -> a -> a
```

Principales classes prédéfinies



Types prédéfinis

Le type trivial noté () désigne le vide ou "void", dérive de Ix, Enum, Read, Show, Bounded.

Le type booléen Bool n'a que deux valeurs False | True, dérive des mêmes et a les opérations &&, ||, not ... Le type Char dérive encore des mêmes classes et on a les fonctions isAscii, isSpace, isUpper, isLower, isAlpha, isDigit ...

Calcul automatique des types et contraintes de classes

Dérivation numérique et résolution d'une équation par la méthode de Newton :

```

deriv f dx = \ x -> (f(x + dx) - f(x)) / dx
deriv :: Fractional a => (a -> a) -> a -> a -> a
Exemple > deriv (\ x -> x*x*x) 0.0001 1 3.00030001
newton f x dx eps = if abs (f x) < eps then x else newton f (x - (f x) / (f' x)) dx eps where f' = deriv f dx
newton :: (Fractional a, Ord a) => (a -> a) -> a -> a -> a -> a
Exemple > 2 * (newton cos 1 0.001 0.0001) 3.14159265

```

Types définis par constructeurs

Le mot réservé "data" sert à définir les types (voir plus loin "type"). On peut définir un point d'éléments de types a par un seul constructeur Pt à deux champs, après quoi on définit deux fonctions abscisse et ordonnée, ou bien avec la définition plus classique de "record".

```

data Point a = Pt a a
absi (Pt x _) = x
ordo (Pt _ y) = y
pn = Pt 2 3      est de type Point Integer      pb = Pt True False      est de type Point Bool
modul (Pt x y) = sqrt(x*x + y*y)
modul :: Floating a => Point a -> a

```

Ceci correspond complètement avec la définition classique de "record" avec "champs", sauf que le type doit y être précisé:

```
data Point = Pt {absi, ordo :: Float}
```

Types définis par énumération

Tandis que les constructions d'expressions n'interviennent qu'à l'exécution, l'homogénéité des constructions de type est vérifiée à la compilation.

```
data Color = Rouge | Jaune | Vert | Bleu
Le constructeur de listes est prédéfini par récurrence par :
data [a] = [] | a:[a]
```

Exemple du jour de la semaine

Le jour de la semaine correspondant à une date, suivant la formule de Zeller : si le mois m est supérieur ou égal à 3 on le change en m-2 et sinon en m+10 ainsi que l'année a en a-1. On pose alors na le numéro de l'année dans le siècle et s le numéro du siècle, et enfin f = j + na - 2*s + na div 4 + s div 4 + (26*m - 2) div 10. Le code est donné par f mod 7 (0 si dimanche). De plus, Grégoire XIII a réformé le calendrier le jeudi 4 octobre 1582 qui fut donc suivi du vendredi 15 octobre 1582. Avant ce changement, on ajoute 3 à f.

```

data Jour = Dimanche | Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi deriving (Enum, Show)
jour j m a = let (m', a') = if 3 < m then (m-2, a) else (m+10, a-1)
              in let (s, as, f) = (div a' 100, mod a' 100, j + as - 2*s + (div as 4) + (div s 4) + (div (26*m' - 2) 10))
              in (iterate succ Dimanche) !! (mod f 7)
jour 14 7 1789  Mardi

```

Types définis synonymie

Pour commodité, et sans définir de nouveau type, on peut en nommer certains avec le mot réservé "type". C'est ce qui est prédéfini avec String = [Char]. Le mot "newtype" permet de créer une nouvelle instance avec la même représentation qu'un autre type mais sans que ce soit un synonyme. Ainsi, si on veut quelques différences, on peut définir :

```

newtype Entier = Contenu Integer
toentier n = if n < 0 then error "Un entier est positif." else Contenu n

```

Exemple des arbres binaires étiquetés

```

data Arb a = Feuille a | Noeud a (Arb a) (Arb a) -- arbre binaire étiqueté, les champs ainsi définis sont des fonctions :
:type Feuille  Feuille :: a -> Arb a      :type Noeud  Noeud :: a -> Arb a -> Arb a -> Arb a
parcours (Feuille x) = [x] -- parcours racine-gauche-droite
parcours (Noeud n g d) = n : (parcours g) ++ (parcours d)
voirarb (Feuille x) = show x
voirarb (Noeud x g d) = "<" ++ voirarb g ++ "|" ++ show x ++ "|" ++ voirarb d ++ ">"
Exemple > a = Noeud 0 (Noeud 1 (Feuille 3) (Feuille 4)) (Feuille 2) -- est un petit exemple
parcours a  [0,1,3,4,2]      voirarb a  "<<3|1|4>|0|2>"

```

Exemple du type arbre irrégulier

```

data Arbo a = Nd a [Arbo a]
sommet (Nd x _) = x
fils (Nd _ f) = f
parcours (Nd x []) = [x] -- parcours racine gauche droite
parcours (Nd n q) = n : (foldr (++) [] (map parcours q))

```

```

voirarb (Nd n []) = show n
voirarb (Nd n q) = show n ++ "<" ++ (foldr (++) [] (map voirarb q)) ++ ">"
Exemple > a = Nd 0 [Nd 1 [Nd 3 [], Nd 4 [], Nd 7 []], Nd 2 [Nd 8 [], Nd 5 []]
(sommet.head.fils) a 1
parcours a [0,1,3,4,7,2,8,5]
voirarb a "0<1<347>2<8>5>"

```

Exemple de définition du type binaire

On définit les "Binary" comme listes de "bits" puis les fonctions booléennes :

```

data Bit = O | I deriving (Show, Eq)
type Binary = [Bit]
non [] = []
non (x:q) = (if x == O then I else O) : (non q)
non [O, O, I, O, I, I] [I,I,O,I,O,O]
et [] [] = []
et (x:p) (y:q) = (if x==I && y==I then I else O) : (et p q)
et [O,I,I,O] [I,I,O,O] [O,I,O,O]
ajuste b1 b2 = aj (reverse b1) (reverse b2) [] [] where aj [] [] r1 r2 = (r1, r2)
aj [] (y:q) r1 r2 = aj [] q (O:r1) (y:r2)
aj (x:p) [] r1 r2 = aj p [] (x:r1) (O:r2)
aj (x:p) (y:q) r1 r2 = aj p q (x:r1) (y:r2)
ajuste [I,O,I] [I,I,I,O,O] ([O,O,O,I,O,I],[I,I,I,O,O])
decimal b = dec (reverse b) 1 0 where dec [] _ r = r; dec (x:q) f r = dec q (2*f) (r + (if x == I then f else 0))
decimal [I,O,O,I] 9
binaire n = reverse (binr n) -- binaire fait l'inverse
where binr n = if n == 0 then [] else (if mod n 2 == 0 then O else I):(binr (div n 2))
verif x y = decimal (somb (binaire x) (binaire y))
somb b1 b2 = sb (reverse b1) (reverse b2) [] 0 -- somme binaire
where sb [] [] res 1 = I:res;
sb [] [] res 0 = res;
sb b [] res ret = sb [] b res ret;
sb [] (O:q) res 1 = sb [] q (I:res) 0;
sb [] (I:q) res 1 = sb [] q (O:res) 1;
sb [] (x:q) res 0 = sb [] q (x:res) 0;
sb (x:p) (y:q) res ret = sb p q ((if (mod c 2) == 1 then I else O):res) (div c 2)
where c = (val x) + (val y) + ret where {val O = 0; val I = 1}
somb [I,O,O,O] [I,O,I,O] [I,O,O,I,O] en effet 8+4=12
Compression en débutant par I, ex compressé [I,I,O,I,O,O,O,O,I,I,O,I,I,I] [2,1,1,4,2,1,4]
compressé b = if b == [O] then [0] else let comp' (x:q) = comp q (x==O) 1 [] in comp' (reverse b)
where comp (O:q) True n res = comp q True (n+1) res;
comp (I:q) True n res = comp q False 1 (n:res);
comp (I:q) False n res = comp q False (n+1) res;
comp (O:q) False n res = comp q True 1 (n:res);
comp [] False n res = n:res

```

Décompression et chaîne de caractère issue, par ex [O,I,I,O,I,O,O,O,O,I,I,O,I,I,I,O,I,I,O,I,I,O,O,O,I,I,O,O,O,O,I] "hola" et l'inverse en exercice.

Programmation par actions

Entrée-sorties : une méthode show permet d'afficher tous les objets de la classe Show en renvoyant un String :

essai x y = "La somme de " ++ show x ++ " et " ++ show y ++ " est " ++ show (x+y) ++ "."

Le constructeur d'actions ou effets de bord est IO, on peut donc construire les types IO a pour chaque type a et même des listes d'actions [IO a]

Le type de print est Show a => a -> IO (), ainsi print ("La valeur est " ++ show 5) peut être utilisé.

Outre "print", on dispose de fonctions prédéfinies getLine :: IO String, attend une chaîne, putStr :: String -> IO() affiche une chaîne, writeFile, appendFile :: String -> String -> IO(), readFile :: String -> IO String

GetLine peut être redéfini par : getLine = do c <- getChar; if c == '\n' then return "" else do s <- getLine; return (c:s)

essai = do c <- getChar; putChar c)

essai bis = do c <- getChar; if c == '\n' then return "" else do l <- getLine; return (c:l)

essaiter = [putChar 'a', do putChar 'b'; putChar 'c', do c <- getChar; putChar c]

suite [] = return()

suite (x:q) = do x; suite q

chaine s = suite (map putChar s)
Sont de types respectifs IO(), IO String, [IO()], [IO()] -> IO() et String -> IO()

Exemple d'ouverture et d'écriture de fichiers

```
transfo = do      {putStr "Fichier d'entrée ? "; f <- getLine;
                  putStr "Fichier de sortie ? "; g <- getLine;
                  c <- readFile f; writeFile g (map toUpper c); putStr "Fini\n" }
```

Application aux tours de Hanoï

```
hanoi n = hanoi' n "gauche" "milieu" "droite"      -- Départ, Intermédiaire, Arrivée   hanoi :: Num a => a -> IO ()
  where hanoi' n dep interm arr =
        if n == 0 then return () else do          {hanoi' (n-1) dep arr interm;
                                                    print ("Transfert de " ++ dep ++ " vers " ++ arr);
                                                    hanoi' (n-1) interm dep arr }
```

```
hanoi 3  "Transfert de gauche vers droite"
      "Transfert de gauche vers milieu"
      "Transfert de droite vers milieu"
      "Transfert de gauche vers droite"
      "Transfert de milieu vers gauche"
      "Transfert de milieu vers droite"
      "Transfert de gauche vers droite"
```

Exemple de la suite de Syracuse

Partant d'un entier quelconque, chaque terme est calculé comme la moitié du précédent si celui-ci est pair, ou sinon comme le successeur du triple. Pour ce genre de suites, on observe sans l'avoir démontré que cette suite parvient toujours à la période 1 - 4 - 2 - 1 ... Attention, l'expression "if even n then n/2 else 3*n+1" provoque une surcharge non résolue, le résultat de "if" doit être un Int.

```
suite n = do print (show n ++ " "); if n == 1 then return () else suite (if even n then div n 2 else 3*n+1)
```

Mais on a mieux avec :

```
syr n = takeWhile (\x -> 1 < x) (iterate (\x -> if even x then div x 2 else 3*x+1) n)
```

```
syr 7  [7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2]
```

Exceptions

Grâce au type prédéfini IOError, une exception a le type IOError -> IO a, on dispose des méthodes

```
fail :: IOError -> IO a
```

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

```
error :: String -> a
```

La classe Monad

Un monoïde est une structure algébrique avec une loi binaire interne associative unitaire, exemples, (N, +, 0), (A*, concat, motvide), (F(E, E), composition, I_E), (Instructions, séquence, instrucvide).

Une monade est un constructeur de classe (un opérateur m dans l'algèbre des types comme [], IO ou Maybe), représentant un monoïde et une instance de Functor, est un foncteur entre monoïdes. Les opérateurs d'une monade sont >>= et "return" qui est son neutre. La classe MonadPlus possède en plus un opérateur "mplus" et un "mzero" neutre pour mplus et absorbant pour >>+.m est un opérateur dans l'algèbre des types comme [], IO ou Maybe,

```
Class Monad m where  (>>=) :: m a -> (a -> m b) -> m b
```

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

```
                    return :: a -> m a
```

```
                    fail :: String -> m a
```

En fait e >> e' n'est que la définition e >>= _ -> e', et e >>= (\x -> e') s'écrit désormais do {x <- e; e'} qui signifie le calcul de f(x) où x est le résultat du calcul de e.

Des exemples simples de monades sont :

- La monade liste avec return x = [x] et q >>= f = concat (map f q), aussi instance de MonadPlus avec mzero = [] et mplus = (++) . Pour les listes, il faut remarquer qu'on retrouve l'écriture do {a <- [1,2,3]; b <- [3,4,5]; return (a+b)} avec [a+b | a <- [1,2,3], b <- [4,5,6]]

- La monade Maybe modélise les exceptions, return s'y nomme Just, data Maybe x = Just x | Nothing

- La monade entrées-sortiesm

La classe Functor possède la méthode fmap, elle permet de définir des constructeurs de types sur toute une classe de types, avec un exemple :

```
Class Functor f where fmap :: (a -> b) -> f a -> f b
```

```
instance Functor Arb where      fmap (Feuille x) = feuille (f x)
```

```
fmap f (Noeud g d) = Noeud (fmap g d) (fmap f d)
```

Le type structuré Arb (qui dépend d'un type a) devient ainsi une instance de Functor.

Tableaux

Dans l'idéal, les matrices seraient des fonctions (des suites doubles presque nulles) mais une telle implémentation est inefficace, aussi tout est défini dans un module Array utilisant la classe Ix des "index". La création d'un tableau se fait grâce à la fonction "array" et l'accès à un élément par ! comme pour les listes qu'ils sont. a // [(i,v)] désigne la matrice a où a!i est remplacé par v, exemple de swap où deux lignes sont échangées.

```
import Array -- module te-ableaux, il en existe d'autres à propos des complexes, piles ...
```

```
cubes = array (1, 100) [(i, i*i*i) | i <- [1..100]]
```

```
cubes ! 3 27
```

```
fib n = a where a = array (0, n) [(0,1), (1,1)] ++ [(i, a!(i-2) + a!(i-1)) | i <- [2..n]]
```

```
fib 12 [0,1,1,2,3,5,8,13,21,34,55,89,144,233]
```

```
pascal n = a where a = array ((0,0), (n,n))
```

```
[(i,j), (i,j)] ++ [(i,j), a!(i-1,j-1) + a!(i-1,j) | j <- [1..n], i <- [1..n]]
```

```
swap i i' a = a // [nouv | j <- [jinf, jsup], nouv <- [(i,j), a!(i',j)], (i',j), a!(i,j)]]
```

```
where ((iinf, jinf), (isup, jsup)) = bounds a
```

```
transpose a = array ((lj,uj), (li,ui)) [a!(j,i) | i <- [li,ui], j <- [lj,uj]]
```

```
where ((li,ui), (lj,uj)) = bounds a
```

```
mult a b = array (if (lj,uj) == (li',ui') then ((li,lj'), (ui,uj')) else error "bornes incompatibles")
```

```
[(i,j), sum [a!(i,k) * b!(k,j) | k <- range (li,uj)]] | i <- [li..ui], j <- [lj'..uj']
```

```
where ((li,lj), (ui,uj)) = bounds a; ((li',lj'), (ui',uj')) = bounds b
```

Exercices

1 Donner le résultat de l'évaluation

```
if 2 > 9 then "oui" else "non"
```

```
let x = (sqrt 16) in (x + 1)
```

```
1:2:3
```

```
[1,2,3] ++ [4]
```

```
1:(2:(3:[4]))
```

```
head [1,2,3]
```

```
tail [1,2,3]
```

```
drop 4 [1,2,3,4,5,6]
```

```
[1,2,3,4] !! 4
```

```
[(x, y) | x <- [1..2], y <- [2..5], (x+y) /= 4]
```

```
[x | x <- [1..10], (x 'mod' 2) == 0]
```

```
map fst [(1,2),(3,8),(0,6),(3,1)]
```

```
(foldr f 0 q, fold f 0 q) where q = [6,9,8,3,10]; f x y = (x+y) 'div' 2
```

```
foldr (++) [] [[1,2,3], [4,5,6], [], [7]]
```

2 Construire une fonction telle que :

Qui compte le nombre d'éléments négatifs d'une liste, en utilisant une définition par compréhension

Calculant la moyenne d'une liste de nombres et qui soit récursive terminale

Qui renvoie l'élément médian d'une liste ayant un nombre impair d'éléments

Qui répète 1 une fois, 2 deux fois etc, dans une liste jusqu'à l'argument n

Qui convertit une chaîne numérique comme "1234" en un entier 1234 grâce à la fonction prédéfinie ord qui donne le code ASCII d'un caractère.

3 Définir les listes suivantes avec une définition par compréhension

```
[1,2,3,4,5,6,7,8,10,11,12,13,14,15]
```

```
[2,-3,4,-5,6,-7,8,-9,10,-11]
```

4 Donner le type des expressions suivantes

```
compose f g x = f(g x)
```

Si data T a = C a | R [T a] est donné, quel est le type de R[C(C True), C(C False)] ?

```
cc f g = \ (x, y) -> f( g x y)
```

```
dd x = [1..x]
```

```
pp x = (x, x)
```

```
qq (x:q) = (x, q)
rr q = (a, b) where a = foldr f 0 q; b = foldr f 0 q; f x y = 10*x + y
ss x = x: ss(x+2)
uu (x, y) z = x = z
tt f x = do y <- f x; return (x == y)
```

Trouver un affichage d'arbre binaire en racine-gauche-droite, qui soit récursif terminal de complexité linéaire contrairement à :

```
aff (Feuille x) = show x
aff (Noeud g d) = "(" ++ aff g ++ "," ++ aff d ++ ")"
```