

# Le langage de programmation Haskell

Stéphane Bortzmeyer  
AFNIC  
bortzmeyer@nic.fr

16 janvier 2007

Ce document est distribué sous les termes de la GNU Free Documentation License

<http://www.gnu.org/licenses/licenses.html#FDL>.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation ; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

# Pourquoi parler d'Haskell ?

# Pourquoi parler d'Haskell ?

1. Élargir la perspective : il y a beaucoup de langages mais ils sont souvent proches, membres de la même famille (exemple : Perl, Python et Ruby).

# Pourquoi parler d'Haskell ?

1. Élargir la perspective : il y a beaucoup de langages mais ils sont souvent proches, membres de la même famille (exemple : Perl, Python et Ruby).
2. Parmi les langages « alternatifs », Haskell est un des plus répandus et des plus utilisables.

# Pourquoi parler d'Haskell ?

1. Élargir la perspective : il y a beaucoup de langages mais ils sont souvent proches, membres de la même famille (exemple : Perl, Python et Ruby).
2. Parmi les langages « alternatifs », Haskell est un des plus répandus et des plus utilisables.
3. C'est cool.

Haskell appartient à cette grande famille, où on trouve aussi Lisp, Scheme, la famille ML (comme CAML) . . .

Ces langages se caractérisent techniquement par l'importance donnée aux **fonctions**.

1. Composition de fonctions,
2. Fonctions de première classe (peuvent être passées comme paramètres, renvoyées par une fonction),
3. Algèbre de fonctions (compositions).

Il y a d'autres caractéristiques des langages fonctionnels, bien que non directement liées aux fonctions, notamment l'usage intensif des **listes** et des fonctions sur les listes.

Ces langages se caractérisent socialement par la prédominance de mathématiciens universitaires incompréhensibles et ignorant des réalités du vrai monde où on gagne de l'argent.

## Suite sur les langages fonctionnels

Ces langages se caractérisent socialement par la prédominance de mathématiciens universitaires incompréhensibles et ignorant des réalités du vrai monde où on gagne de l'argent.

Le langage fonctionnel classique est remarquable, admiré et jamais utilisé.

# Petite introduction pratique

- ▶ Qui maintient Haskell ? Pas de réponse simple, c'est un effort communautaire, la norme actuelle est Haskell 98, beaucoup d'extensions sont largement utilisées mais il n'a pas encore été possible de normaliser une nouvelle version.

## Petite introduction pratique

- ▶ Qui maintient Haskell ? Pas de réponse simple, c'est un effort communautaire, la norme actuelle est Haskell 98, beaucoup d'extensions sont largement utilisées mais il n'a pas encore été possible de normaliser une nouvelle version.
- ▶ Mises en œuvre : les deux principales sont Hugs (uniquement interprété) et ghc (Glasgow Haskell Compiler, maintenu chez Microsoft Research, le plus gros logiciel libre financé par Microsoft).

## Petite introduction pratique

- ▶ Qui maintient Haskell ? Pas de réponse simple, c'est un effort communautaire, la norme actuelle est Haskell 98, beaucoup d'extensions sont largement utilisées mais il n'a pas encore été possible de normaliser une nouvelle version.
- ▶ Mises en œuvre : les deux principales sont Hugs (uniquement interprété) et ghc (Glasgow Haskell Compiler, maintenu chez Microsoft Research, le plus gros logiciel libre financé par Microsoft).
- ▶ Communauté : Liste haskell-cafe, canal #haskell sur Freenode, <http://www.haskell.org/> (un Wiki)... Très active et très serviable.

Alors, ça commence ?

Pas encore de *Hello, world?*

```
main = putStrLn "Hello, world"
```

# Commençons vraiment

On part évidemment des fonctions.

```
factorielle 1 = 1  
factorielle n = n * factorielle(n-1)
```

On a défini une fonction, *factorielle*, par *pattern matching*. La factorielle de 1 vaut 1. Les autres sont définies récursivement.

## Une fonction plus riche

```
-- Canonicalize city names, according to AFNIC registration rules :-(
canonicalize cityname =
  filter
    (\c -> if (c == '-' ) || (c == '\\') || (c == ' ') then
      False
    else
      True)
  cityname
```

# Une fonction plus riche

```
-- Canonicalize city names, according to AFNIC registration rules :-(  
canonicalize cityname =  
  filter  
    (\c -> if (c == '-' ) || (c == '\\') || (c == ' ') then  
      False  
    else  
      True)  
  cityname
```

1. filter est prédéfinie (dans le **Prelude**). Elle prend une **fonction** en premier paramètre et un tableau en second.

# Une fonction plus riche

```
-- Canonicalize city names, according to AFNIC registration rules :-(  
canonicalize cityname =  
  filter  
    (\c -> if (c == '-' ) || (c == '\\') || (c == ' ') then  
      False  
    else  
      True)  
  cityname
```

1. filter est prédéfinie (dans le **Prelude**). Elle prend une **fonction** en premier paramètre et un tableau en second.
2. La fonction est une **lambda**, une fonction anonyme. Elle prend un paramètre *c* et envoie un booléen (vrai, je le garde, faux, je le jette).

ghci, interpréteur de ghc, peut nous montrer la fonction en action :

```
*Main> canonicalize "Paris"  
"Paris"  
*Main> canonicalize "L'Abergement-Clemenciat"  
"LAbergementClemenciat"
```

Haskell est **typé**, contrairement à Lisp.

ghci peut nous indiquer le type :

```
*Main> :t canonicalize  
canonicalize :: [Char] -> [Char]
```

canonicalize transforme un tableau (`[]`) de caractères en un autre.

Haskell est **typé**, contrairement à Lisp.

ghci peut nous indiquer le type :

```
*Main> :t canonicalize  
canonicalize :: [Char] -> [Char]
```

canonicalize transforme un tableau (`[]`) de caractères en un autre.

Minute, où ai-je déclaré ce type ?

Haskell est **typé**, contrairement à Lisp.

ghci peut nous indiquer le type :

```
*Main> :t canonicalize  
canonicalize :: [Char] -> [Char]
```

canonicalize transforme un tableau ([]) de caractères en un autre.

Minute, où ai-je déclaré ce type ?

Nulle part, Haskell l'a trouvé tout seul, c'est l'**inférence** de type.

Haskell fait respecter le typage :

```
*Main> canonicalize False
```

```
<interactive>:1:13:
```

```
    Couldn't match expected type '[Char]' against inferred type 'Bool'
```

Si on y tient, on peut déclarer à la main :

```
canonicalize :: String -> String
```

# Signature d'une fonction

Toute fonction a donc une signature : le type des paramètres et celui du résultat. Cette signature peut être **inférée**.

Haskell s'assure que les appels de fonctions correspondent à cette signature.

La signature et le typage permettent au génial logiciel **Hoople** de chercher une fonction par signature.

Supposons qu'on ait une liste de chaînes de caractères et qu'on veuille les concaténer en une seule. On cherche donc une fonction capable de faire “[String] -> String”. On tape cette signature dans Hoople et on a :

```
Prelude.      unlines :: [String] -> String
Prelude.      unwords :: [String] -> String
```

## Et avec plusieurs paramètres ?

Prenons une fonction qui compare deux noms de domaine sous forme canonique

```
(==) :: CityName -> CityName -> Bool
(==) city1 city2 =
    map toUpper (canonicalize city1) ==
    map toUpper (canonicalize city2)
```

```
Prelude Canonicalize> "paris" == "pa - ris"
True
```

On dit que la fonction s'écrit sous forme **curryfiée** (un argument puis un autre, puis un autre...).

Cela permet des **applications partielles** :

```
Prelude Canonicalize> let commeParis = (==) "paris"  
Prelude Canonicalize> commeParis "pa - ris"  
True  
Prelude Canonicalize> commeParis "pari"  
False
```

# Opérations sur les listes

Haskell utilise intensivement les listes (ou tableaux). Plusieurs fonctions sont conçues pour opérer sur les listes.

```
[Prelude> [1, 2, 3]  
[1,2,3]
```

```
Prelude> map (+1) [1, 2, 3]  
[2,3,4]
```

```
Prelude> foldl (+) 0 [1, 2, 3]  
6
```

- ▶ map applique une fonction unaire (ici, “additionner 1”) à tous les éléments de la liste. La signature de map est  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ .

- ▶ `map` applique une fonction unaire (ici, “additionner 1”) à tous les éléments de la liste. La signature de `map` est  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ .
- ▶ `foldl` (*fold from left*) applique une fonction binaire à tous les éléments de la liste en les réduisant à un scalaire. (Il faut indiquer la valeur de départ, ici zéro.) La signature de `foldl` est  $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$ .

foldl est appelé reduce dans l'article historique *MapReduce : Simplified Data Processing on Large Clusters* qui a popularisé ces deux opérations, en leur attribuant tout le succès de Google (<http://labs.google.com/papers/mapreduce-osdi04.pdf>).

foldl est appelé reduce dans l'article historique *MapReduce : Simplified Data Processing on Large Clusters* qui a popularisé ces deux opérations, en leur attribuant tout le succès de Google (<http://labs.google.com/papers/mapreduce-osdi04.pdf>).

## Plus besoin de boucles

Les boucles, comme for, structure de bas niveau pour traiter les listes, sont absentes de Haskell. Map/Reduce et la récursivité - dans de rares cas - suffisent à les remplacer.

## Évaluation paresseuse

L'évaluation en Haskell est **paresseuse** (*lazy*). Les valeurs ne sont évaluées que si nécessaires.

```
crash =
    error "I crash and burn"

result = (elem 'o' "foo") || crash

main = do
    putStrLn "I start"
    putStrLn (show result)
    putStrLn (show crash)
```

Le programme ne plantera qu'à la troisième ligne. (`||`), *or* est paresseux.

# La paresse permet de conquérir l'infini

L'évaluation paresseuse permet des structures de données infinies.

Cas réel : un évaluateur d'un langage de grammaire formelle, ABNF (RFC 4234).

```
data Production = Choice [Production] | Sequence [Production] |  
                  Repeat Int Int Production | Constant String
```

Une production de la grammaire peut contenir d'autres productions qui, à leur tour... Il n'y a pas forcément de condition d'arrêt (cas de la syntaxe Lisp).

Sans l'évaluation paresseuse, de telles structures récursives ne seraient pas possibles.

## Pur comme l'agneau

Presque tous les langages fonctionnels sont **impurs** : ils permettent des effets de bord, comme les entrées-sorties ou comme les affectations.

Leurs fonctions ne sont donc pas des vraies fonctions au sens mathématique.

Cela a des conséquences pratiques. Par exemple :

```
x = f1() + f1()  
/* Est-ce que le compilateur a le droit de reecrire  
cela en 2 * f1() ? */
```

# Haskell est pur

Les fonctions ne peuvent pas faire d'effet de bord. Voir plus loin les monades pour un échappement contrôlé vers le monde réel.

Une autre conséquence de la pureté est de permettre la génération automatique de tests d'une fonction.

Le module QuickCheck fait cela, en générant aléatoirement des paramètres et en vérifiant que le résultat obéit à des propriétés définies par le programmeur.

```

-- Canonicalization is idempotent
prop_idempotent s = (canonicalize . canonicalize) s ==
                    canonicalize s
    where _ = s :: CityName

-- Canonicalization removes some characters
prop_nomoredashes s = not (elem '-' (canonicalize s)) ||
                      not (elem '\' (canonicalize s))

tests = [("Idempotence", test prop_idempotent),
         ("Delete all dashes", test prop_nomoredashes)]

...

% runhaskell Tests.hs
Idempotence           : OK, passed 100 tests.
Delete all dashes     : OK, passed 100 tests.

```

Les E/S sont des effets de bord. Donc, un langage pur ne doit pas avoir d'E/S.

Les E/S sont des effets de bord. Donc, un langage pur ne doit pas avoir d'E/S.

OK, je plaisantais. Un langage sans E/S est peu utile.

Les E/S sont des effets de bord. Donc, un langage pur ne doit pas avoir d'E/S.

OK, je plaisantais. Un langage sans E/S est peu utile.

Haskell permet des E/S via le concept de **monade**. Le concept est très riche et très abstrait mais on va commencer par une définition simple.

# Monade

Une monade est une structure qui stocke un calcul, comme un **état**, par exemple, pour la monade IO, l'état du monde extérieur.

Une fonction qui fait des E/S est donc dans la monade IO et **ne peut pas en sortir** pour garantir la séparation du code pur et du code impur.

```
pure i = 2 * i
```

```
impure i = putStrLn (show i)
```

```
main = let i = 3 in  
        putStrLn (show (pure i)) >> impure i
```

(» veut dire “suivi de” et permet de faire du séquençage d'actions.)

On peut appeler la fonction pure depuis du code qui est dans la monade IO.

L'inverse n'est pas vrai.

# Sucre syntaxique

La façon normale de faire des E/S en Haskell est donc :

```
main = putStrLn "What is your name?" >> getLine >>= putStrLn
```

( $\gg=$  transmet le résultat de la première action, pas  $\gg$ .)

Mais c'est souvent un peu pénible donc Haskell fournit une autre syntaxe (qui a exactement la même signification), la notation `do` :

```
main = do
  putStr "What is your name? "
  name <- getLine
  putStrLn ("Hello, " ++ name)
```

## Le sucre syntaxique, c'est bon

Cette syntaxe permet de faire facilement de l'impératif en Haskell.

## Dans le monde réel : whois

```
import Network
import IO
import System
port = Service "whois"
main = do
    myargs <- getArgs
    let
        host = myargs !! 0
        query = myargs !! 1
    h <- connectTo host port
    hSetBuffering h LineBuffering
    hPutStr h (query ++ "\r\n")
    response <- hGetContents h
    putStrLn response
```

import permet d'importer d'autres modules. Ici, uniquement des modules standard de ghc.

Les caractéristiques d'Haskell permettent d'aborder certains problèmes de manière nouvelle.

Par exemple, l'analyse syntaxique : plus de préprocesseur, comme Yacc, tout est dans le langage.

```
-- named.conf
comment = do
  (string "/" <|> string "#")
  many (noneOf "\n\r")
  crlf
  <?> "Comment"

crlf = do {string "\n"; optional (string "\r")} <?> "End of line"
```

Ici, on définit une production comment, composée des caractères de début, et terminée par `crlf`. `many`, `string` et `noneOf` sont fournis par la bibliothèque `Parsec`.

## Exemple d'analyse : les *language tags*

Les *language tags* du RFC 4646 ne peuvent pas se décrire facilement par une regexp. Un vrai analyseur est nécessaire. (Le code est extrait de GaBuZoMeu, développé à l'AFNIC.)

```
region      = do
    count 2 letter      -- ISO 3166 code
  <|> count 3 digit    -- United Nations M.49 code

privateuse = do
  charX
  values <- many1 (do {dash; countBetween 1 8 alphaNum})
  eof
  return (Priv values)
```

darcs est un VCS (contrôle de version) décentralisé, le premier à avoir été utilisable.

L'auteur a publié une bonne analyse des avantages et inconvénients d'Haskell pour un vrai programme :

[http://darcs.net/hw\\_2005.pdf](http://darcs.net/hw_2005.pdf). Lire aussi

[http://darcs.net/cufp\\_2005.pdf](http://darcs.net/cufp_2005.pdf) et

<http://osdir.com/Article2571.phtml>.

Première (et jusqu'à présent unique) mise en œuvre de Perl 6. (Le langage avec la grammaire modifiable en vol.)

## Gros programmes réels : pugs

Première (et jusqu'à présent unique) mise en œuvre de Perl 6. (Le langage avec la grammaire modifiable en vol.)

Écrit par Atrijus Tang (non, Audrey Tang, pugs est le seul programme dont l'auteur a changé de sexe avant la 1.0).

Son cours Haskell :

<http://feather.perl6.nl/~audreyt/osdc/haskell.xul>

Les types sont organisés en classes.

```
next n = n + 1
```

```
...
```

```
*Main> :t next
```

```
next :: (Num a) => a -> a
```

Num est une classe de types qui comprend tous les types numériques. Le début de la signature se lit donc “Pour tout type a qui appartient à la classe Num” ...

## Plus tard

Monades (dans leur généralité) et flèches ne sont pas couvertes par la version GFDL de cet exposé.

Monades (dans leur généralité) et flèches ne sont pas couvertes par la version GFDL de cet exposé.

Écrivez à l'auteur pour un devis.