Une introduction agréable au langage Haskell 98

Paul Hudak (Yale University),
John Peterson (Yale University)
et Joseph Fasel (Los Alamos National Laboratory)

Traduction par: Nicolas Vallée

Il s'agit d'une traduction d'un ouvrage de référence sur Haskell écrit par : Paul Hudak (Yale University), John Peterson (Yale University) et Joseph Fasel (Los Alamos National Laboratory) Vous y trouverez une présentation de toutes les grandes caractéristiques du langage Haskell. Attention, il est toutefois conseillé d'avoir de bonnes bases de programmation fonctionnelle avant de se lancer dans sa lecture.

Nous tenons à remercier Gnux, ggnore, fearyourself, Heureux-oli, Kikof et khayyam90 pour leurs soutiens et leurs relectures. Lecture, correction finale par Dan V.Popa (user: Ha$kell) .

Version PDF par Dan V. Popa (Univ. Bacău) initiateur du projet Haskell-fr (Communauté française des utilisateurs de langage Haskell – à voir la page web http://www.haskell.org/haskellwiki/Fr/Haskell) v.0.2

le 23 mai 2007 Ha$kell

A Gentle Introduction to Haskell 98

Copyright (C) 1999 Paul Hudak, John Peterson and Joseph Fasel. Permission is hereby granted, free of charge, to any person obtaining a copy of "A Gentle Introduction to Haskell" (the Text), to deal in the Text without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Text, and to permit persons to whom the Text is furnished to do so, subject to the following condition: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Text.

Introduction
I. Introduction 7

Valeurs, types, et autres friandises 9
II. Valeurs, types, et autres friandises 9
II-1. Les types polymorphes 11
II-2. Types définis par l'utilisateur 13
II-2-1. Les types récursifs 14
II-3. Les synonymes de Type 15
II-2-4. Les types internes n'ont rien de particulier 16
II-4-1. Les compréhensions de listes et les séquences arithmétiques 18
II-4-2. Chaînes de caractères 18

Les Fonctions 20
III. Les Fonctions 20
III-1. Les abstractions lambda 21
III-2. Opérateurs infixes 21
III-2-1. Sections 21
III-2-2. Les déclarations de fixité 22
III-3. Les fonctions sont non-strictes 23
III-4. Les structures de données "infinies" 24
III-5. La fonction d'erreur 25

Les Expressions Case et la correspondance de motifs 27
IV. Les Expressions Case et la correspondance de motifs 27
IV-1. La sémantique des correspondances de motifs 29
IV-2. Un exemple 29
IV-3. Les expressions casuelles (Case expressions) 30
IV-4. Les motifs paresseux (Lazy-patterns) 31
IV-5. Cadrage lexical et formes emboîtées 33
IV-6. Mise en forme 35

Les Classes de types et la surcharge 36
V. Les classes de types et la surcharge 36

Les types, encore 43
VI. Les types, encore 43
VI-1. La déclaration newtype 43
VI-2. Les étiquettes de champs 44
VI-3. Les constructeurs stricts de données 45

Entrées/Sorties 48
VII. Entrées/Sorties 48
VII-1. Opérations d'E/S de base 48

VII-2. Programmer avec des actions 50
VII-3. Gestion des exceptions 52
VII-4. Fichiers, canaux et gestionnaires 53
VII-5. Haskell et la programmation impérative 54

Les Classes standards de Haskell 56
VIII. Les Classes standards de Haskell 56
VIII-1. Classes pour l'égalité et la relation d'ordre 56
VIII-2. La Classe Enumeration 56
VIII-3. Les Classes Read et Show 57
VIII-4. Les Instances dérivées 59

Les Monades 63
IX. A propos des monades 63
IX-1. Les classes monadiques 63
IX-2. Les monades intégrées 66
IX-3. Utilisation des monades 67

Les Nombres 73
X. Les Nombres 73
X-1. Structure des classes numériques 73
X-2. Les Nombres construits 74
X-3. Les Conversions numériques et les surcharges de littéraux 75
X-4. Les Types numériques par défaut 76

Les Modules 78
XI. Les Modules 78
XI-1. Les Noms qualifiés 79
XI-2. Les Types de données abstraits 80
XI-3. Plus de caractéristiques 80

Les Pièges du typage 82
XII. Pièges du typage 82
XII-1. Let-Bound Polymorphism 82
XII-2. Surcharge numérique 82
XII-3. Les Restrictions monomorphiques 83

Les Tableaux 84
XIII. Les Tableaux 84
XIII-1. Les types d'indice 85
XIII-2. Création d'un tableau 85
XIII-3. Accumulation 87
XIII-4. Mises à jour incrémentales 87
XIII-5. Un exemple : la multiplication matricielle 88

Conclusion **90**
XIV. Conclusion 90
XIV-1. Prochaine étape 90
XIV-2. Remerciements 90

Bibliographie **91**

I. Introduction

L'objectif de ce tutoriel n'est pas d'apprendre à programmer, ni même d'apprendre la programmation fonctionnelle. Ca doit plutôt être un supplément au Haskell Report [4], qui fait, de toute manière, une présentation très complète et très technique. Notre objectif est de faire une légère introduction à Haskell pour quelqu'un qui a déjà une expérience avec au moins un autre langage, de préférence un langage fonctionnel (même si ce n'est qu'un langage "quasiment fonctionnel" comme ML ou Scheme). Si le lecteur souhaite en apprendre davantage au sujet de la programmation fonctionnelle, nous recommandons fortement l'ouvrage de Bird Introduction to Functional Programming [1] ou bien celui de Davie : An Introduction to Functional Programming Systems Using Haskell [2]. Pour une étude approfondie des langages de programmation fonctionels et des techniques, incluant quelques uns des principes de la théorie des langages, voir [3].

Le langage Haskell a beaucoup évolué depuis sa création en 1987. Ce tutoriel utilisera Haskell 98 [4]. Les anciennes versions de ce langage sont maintenant obsolètes ; les utilisateurs de Haskell sont encouragés à utiliser Haskell 98. Il y a aussi beaucoup d'extensions à Haskell 98 qui ont été implémentées. Elles ne font pas, à proprement parler, partie du langage Haskell et ne sont pas couvertes par ce tutoriel.

La méthode que nous utiliserons pour présenter les caractéristiques du langage est la suivante : intérêt de l'idée, quelques définitions, des exemples, et enfin les liens vers le Haskell Report pour les détails. Nous suggérons aussi au lecteur de ne pas tenir compte des détails avant d'avoir entièrement lu Gentle Introduction. D'un autre côté, la préface du Standard Haskell (en annexe A du Report) et les bibliothèques standard (dans le Library Report [5]) contiennent beaucoup d'exemples utiles de codes Haskell ; nous en recommandons la lecture attentive une fois ce tutoriel terminé. Ca ne va pas seulement donner au lecteur quelques idées sur ce qu'est un réel code Haskell, mais ça va aussi le familiariser avec une série de fonctions et de types prédéfinis qui font partie du standard Haskell.

Et pour finir, le site web Haskell, http://www.haskell.org contient beaucoup d'informations sur le langage Haskell et ses implémentations.

[Nous avons aussi fait le choix de ne pas présenter pléthore de règles de syntaxe lexicale dès le début. Nous préférons les présenter petit à petit, quand les exemples les demanderont. Elles seront présentées entre crochets, tout comme ce paragraphe. Cela tranche franchement avec l'organisation du Report, mais le Report reste la documentation faisant autorité (les références telles que "Report section 2.1" correspondant aux sections du Report).]

Haskell est un langage de programmation fortement typé : (inventé par Luca Cardelli) les types sont "persistants" et un débutant doit être bien conscient dès le début de toute la puissance et de la complexité du système de typage de Haskell. Pour ceux qui ont

principalement une expérience avec des langages "peu typés" tels que Perl, Tcl, ou Scheme, cela va nécessiter une adaptation relativement difficile ; pour ceux qui sont familiers de Java, C, Modula ou encore ML, l'adaptation devrait être facile mais pas triviale, étant donné que le système de typage de Haskell est différent et plus évolué que beaucoup d'autres. Dans tous les cas, la programmation "typée" fait partie de l'expérience de la programmation Haskell et ne peut pas être évitée.

II. Valeurs, types, et autres friandises
II-1. Les types polymorphes
II-2. Types définis par l'utilisateur
II-2-1. Les types récursifs
II-3. Les synonymes de Type
II-2-4. Les types internes n'ont rien de particulier
II-4-1. Les compréhensions de listes et les séquences arithmétiques
II-4-2. Chaînes de caractères

II. Valeurs, types, et autres friandises

Parce qu'Haskell est un langage purement fonctionnel, tous les calculs sont fait via l'évaluation d'expressions (terme syntaxique) qui retournent des valeurs (entités abstraites que l'on considère comme des réponses). À chaque valeur est associé un type (intuitivement, on peut se représenter les types comme des ensembles de valeurs). Les expressions peuvent être des valeurs atomiques telles que l'entier 5, le caractère 'a', ou la fonction \x -> x+1, mais elles peuvent aussi être des valeurs structurées telle que la liste [1,2,3] ou la pair ('b',4).

De même que les expressions dénotent des valeurs, les expressions de type sont des termes syntaxiques qui dénotent des valeurs de type (ou, plus simplement, des types). Les expressions de type peuvent être des types atomiques tels que Integer (l'ensemble infini des entiers), Char (les caractères) ou Integer->Integer (les fonctions qui, à un Interger associent un Integer), mais elles peuvent aussi être des types structurés tel que [Integer] (liste homogène d'entiers) ou (Char,Integer) (les paires composées d'un caractère et d'un entier).

Avec Haskell, toutes les valeurs sont dites de « première-classe », c'est à dire qu'elles peuvent être passées à des fonctions en tant qu'arguments, retournées en tant que résultats, placées dans des structures de données, etc. En revanche, les types ne sont pas de « première-classe ». En un sens, les types décrivent des valeurs; On appelle typage, l'association d'une valeur avec son type. En reprenant les exemples donnés plus haut, nous écririons leur typage comme suit :

```
      5   :: Integer
    'a'  :: Char
    inc  :: Integer -> Integer
[1,2,3]  :: [Integer]
('b',4)  :: (Char,Integer)
```

Les deux points (« :: ») se lisent « est de type ».

Avec Haskell, les fonctions sont en principe définies par une série d'équations. Par exemple, la fonction inc peut être définie par la simple équation :

```
inc n          = n+1
```

Une équation est un exemple de déclaration. Un autre exemple de déclaration est la signature de type (4.4.1), avec laquelle nous pouvons déclarer un type explicite pour inc :

```
inc             :: Integer -> Integer
```

Nous aurons encore beaucoup à dire à propos de la définition des fonctions dans la section 3.

À des fins pédagogiques, quand nous voudrons indiquer que l'évaluation d'une expression e1 retourne une autre expression ou une valeur e2, nous écrirons :

```
e1 => e2
```

Par exemple, notons que :

inc (inc 3) => 5

Le système de typage statique d'Haskell définit une relation formelle entre des types et des valeurs (4.1.4). Ce système garantit que les programmes Haskell opèrent sur des types cohérents; autrement dit, que le programmeur n'aura pas d'erreurs d'incompatibilité de types. Par exemple, il n'est pas possible, en principe, d'additionner deux caractères; ainsi, l'expression 'a'+'b' est incohérente en raison du type de ses opérandes (en anglais on dit que l'expression est ill-typed). L'avantage principal des langages à types statiques est bien connu : toutes les erreurs de type sont détectées à la compilation. Cependant, un tel système ne peut détecter toutes les erreurs; le type d'une expression telle que 1/0 n'est pas incohérent mais son évaluation provoquera une erreur à l'exécution. Néanmoins, ce système permet de détecter beaucoup d'erreurs à la compilation, d'aider l'utilisateur à raisonner sur les programmes, et permet également au compilateur de générer un code plus efficace (par exemple, les marqueurs ou les tests de type ne sont pas requis à l'exécution).

Ce système de typage garantit également que les signatures de type fournies par l'utilisateur sont correctes. En fait, le système de typage d'Haskell est même suffisamment puissant pour nous épargner l'écriture de toute signature de type (À quelques exceptions près, qui seront décrites plus tard); On dit que le système de typage infère les types corrects. Néanmoins, il est recommandé d'utiliser judicieusement les signatures de type, telles que celles que nous avons données pour inc, parce qu'elles sont une forme efficace de documentation du code et aident à mettre en évidence les erreurs de programmation.

[Le lecteur notera l'emploi d'une majuscule à la première lettre des identificateurs qui dénotent un type spécifique, tels que Integer et Char, mais pas sur les identificateurs qui dénotent des valeurs, tel que inc. Ce n'est pas une simple convention : Haskell impose cette syntaxe lexicale. De plus, la casse des caractères suivants est également importante : foo, fOo et fOO sont tous des identificateurs distincts.]

II-1. Les types polymorphes

Haskell intègre également les types polymorphes, c'est-à-dire des types qui sont universellement quantifiés d'une certaine manière pour tous les types. Les expressions de type polymorphes décrivent essentiellement des familles de types. Par exemple, (forall a)[a] est la famille de types qui comprend, pour tout type a, le type de listes de a. Les listes d'entiers (p. ex. [1,2,3]), les listes de caractères (p. ex. ['a','b','c']), et même les listes de listes d'entiers (p. ex. [[1,2,3],[4,5,6]]), etc. sont toutes membres de cette famille. (Notons, toutefois, que [2,'b'] n'est pas un exemple valide, puisqu'il n'existe pas de type simple contenant à la fois 2 et 'b'.)

[Les identificateurs tel que a ci-dessus sont appelés variables de type, et sont en minuscules pour les distinguer des types spécifiques tel que Int. De plus, puisque Haskell utilise uniquement des types universellement quantifiés, il n'est pas nécessaire d'écrire explicitement le symbole de la quantification universelle, et donc, dans l'exemple ci-dessus, nous écrivons simplement [a]. En d'autres termes, toutes les variables de type utilisent implicitement une quantification universelle.]

Les listes sont des structures de données utilisées fréquemment dans les langages fonctionnels et elles sont un bon véhicule pour expliquer les principes du polymorphisme. Dans Haskell, la liste [1,2,3] est en réalité un raccourci pour la liste 1:(2:(3:[])), où [] est la liste vide et : est l'opérateur infixe qui ajoute son premier argument au début de son deuxième argument (une liste). (: et [] sont, respectivement, l'équivalent de cons et nil dans Lisp.) Puisque : est associatif à droite, nous pouvons également écrire cette liste comme ceci : 1:2:3:[].

Comme exemple d'une fonction définie par l'utilisateur et opérant sur une liste, considérons l'opération qui consiste à compter les éléments contenus dans une liste :

```
length                    :: [a] -> Integer
length []                 =  0
length (x:xs)             =  1 + length xs
```

Cette définition s'explique presque d'elle-même. Nous pouvons lire l'équation comme suit : « La longeur d'une liste vide est 0, et la longeur d'une liste dont le premier élément est x et le reste de la liste est xs est égal à 1 plus la longueur de xs. » (Notons la convention de dénomination utilisée ici ; xs est le pluriel de x, et devrait être lu comme tel.)

Bien qu'il soit intuitif, cet exemple met en lumière un aspect important de Haskell qui doit encore être expliqué : La correspondance de motif (pattern matching en anglais). Le terme gauche de l'équation contient des motifs (patterns) tels que [] et x:xs. Dans l'application d'une fonction, ces motifs sont mis en correspondance avec le paramètre réel d'une manière intuitive ([] correspond uniquement à une liste vide et x:xs correspond à n'importe quelle liste contenant au moins un élément, associant x au premier élément et xs au reste de la liste). Si la correspondance est positive, le terme droit de l'équation est

évalué et retourné en tant que résultat de l'application. Si une correspondance est négative, la correspondance de l'équation suivante est testée, et si toutes les correspondances sont négatives, une erreur en résulte.

La définition de fonctions par correspondance de motifs est très courante avec Haskell, et l'utilisateur devrait se familiariser avec toutes les sortes de motifs qui sont autorisées; Nous reviendrons sur cet aspect dans la section 4.

La fonction length est également un exemple de fonction polymorphe. Elle peut être appliquée à une liste contenant des éléments de n'importe quel type. Par exemple : [Integer], [Char], or [[Integer]].

```
length [1,2,3]       =>   3
length ['a','b','c']      =>   3
length [[1],[2],[3]]      =>   3
```

Voici 2 autres fonctions polymorphes utiles, opérant sur des listes, que nous utiliserons plus tard. La fonction head retourne le premier élément d'une liste, la fonction tail retourne toute la liste à l'exception du premier élément.

```
head                    :: [a] -> a
head (x:xs)             =  x

tail                    :: [a] -> [a]
tail (x:xs)             =  xs
```

Contrairement à length, ces fonctions ne sont pas définies pour toutes les valeurs possibles de leur argument : Une erreur d'exécution surviendra quand ces fonctions seront appliquées à une liste vide.

Formellement, les types polymorphes sont plus généraux que les autres, dans le sens ou l'ensemble de valeurs qu'ils définissent est plus grand. Par exemple, le type [a] est plus général que le type [Char]. En d'autres termes, le deuxième peut être dérivé du premier par une substitution appropriée de a. Concernant cette hiérarchie des généralisations, le système de typage d'Haskell a deux importantes propriétés : Premièrement, toute expression correctement typée est garantie d'avoir un type principal unique (explication plus bas) et deuxièmement, le type principal peut être inféré automatiquement (4.1.4). En comparaison d'un langage mono-morphique tel que C, le lecteur s'apercevra que le polymorphisme améliore l'expressivité, et que l'inférence de type diminue le fardeau du typage supporté par le programmeur.

Le type principal d'une expression ou d'une fonction est le type le moins général qui, intuitivement, « contient toutes les occurrences de l'expression ». Par exemple, le type principal de head est [a]->a; Par contre [b]->a, a->a, ou même a sont des types corrects, mais trop généraux, tandis que [Integer]->Integer est trop spécifique. L'existence d'un type principal unique est une caractéristique du système de typage Hindley-Milner, qui est à la

base du système de typage de Haskell, ML, Miranda, (« Miranda » est une marque déposée de Research Software, Ltd.) et de plusieurs autres langages (principalement fonctionnels).

II-2. Types définis par l'utilisateur

Avec Haskell, nous pouvons définir nos propres types de données en utilisant une déclaration data (4.2.1) dont voici une introduction par une série d'exemples :

Dans Haskell, un type prédéfini important est celui des valeurs de vérité :

```
data Bool                = False | True
```

Le type ainsi défini est le type Bool, et il a exactement deux valeurs : True et False. Le type Bool est un exemple de constructeur de type (nulaire), et True et False sont des constructeurs de données (ou simplement constructeurs).

Dans le même esprit, nous pourrions avoir besoin de définir un type « couleur » :

```
data Color               = Red | Green | Blue | Indigo | Violet
```

Les types Bool et Couleur sont tous deux des exemples de types énumératifs, puisqu'ils consistent en une série finie de constructeurs (nulaires) de données.

Voici un exemple de type comprenant uniquement un constructeur de données :

```
data Point a             = Pt a a
```

Parce qu'il a un unique constructeur, un type comme Point est souvent appelé un type tuple, puisqu'il est essentiellement le produit cartésien (binaire, dans ce cas) d'autres types. (Les tuples correspondent plus-ou-moins à des records (enregistrements) dans d'autres langages.) En revanche, les types à constructeurs multiples, tels que Bool et Couleur, sont appelés des types union (disjoint) ou somme.

Plus important encore, Point est un exemple de type polymorphe : pour tout type t, il définit le type des points cartésiens utilisant t comme type de coordonnées. Le type Point peut maintenant clairement être vu comme un constructeur de type unaire, puisqu'à partir du type t il construit un nouveau type Pointt. (Dans ce sens, l'exemple de liste donné plus haut, [], est également un constructeur de type. À n'importe quel type t donné, on peut « appliquer » [] qui retourne un nouveau type [t]. La syntaxe d'Haskell permet d'écrire []t de cette manière : [t]. De la même manière, -> est un constructeur de type : Soit les types donnés t et u, t->u est le type des fonctions associant des éléments de type t à des éléments de type u.)

Notons que le type du constructeur de données Pt est a -> a -> Point a, et par conséquent,

les typages suivants sont valides :

```
Pt  2.0  3.0            :: Point Float
Pt  'a'  'b'            :: Point Char
Pt True False           :: Point Bool
```

Par contre, une expression telle que Pt 'a' 1 est mal typée puisque 'a' et 1 sont de types différents.

Il est important de faire une distinction entre l'application d'un constructeur de données pour obtenir une valeur et l'application d'un constructeur de type pour obtenir un type; La première se produit à l'exécution et c'est de cette manière que les choses sont calculées dans Haskell, alors que la deuxième se produit à la compilation et fait partie du système de typage qui garantit la validité des types.

[Les constructeurs de type tel que Point et les constructeurs de données, tels que Pt sont chargés dans des espaces de noms différents. Cela permet d'utiliser le même nom pour un constructeur de type et un constructeur de données, comme le montre l'exemple suivant :

```
data Point a = Point a a
```

Même si cela peut prêter à confusion, cela permet de mettre en évidence le lien entre un type et son constructeur de données.]

II-2-1. Les types récursifs

Les types peuvent aussi être récursifs, comme dans le cas du type des arborescences binaires :

```
data Tree a             = Leaf a | Branch (Tree a) (Tree a)
```

Nous avons défini ainsi un type polymorphe d'arborescence binaire dont les éléments sont soit des noeuds « feuille » contenant une valeur de type a, soit des noeuds internes (« branches ») contenant (récursivement) deux sous-arborescences.

À la lecture de telles déclarations de données, il faut se souvenir que Arborescence est un constructeur de type, alors que Branche et Feuille sont des constructeurs de données. Hormis la connexion établie entre ces deux constructeurs, la déclaration ci-dessus définit essentiellement les types suivants pour Branche et Feuille :

```
Branch                  :: Tree a -> Tree a -> Tree a
Leaf                    :: a -> Tree a
```

Avec cet exemple, nous avons défini un type suffisamment riche pour permettre la définition d'intéressantes fonctions (récursives) qui l'utilise. Supposons, par exemple, que nous souhaitions définir une fonction frange qui retourne une liste de tous les éléments

dans les feuilles d'une arborescence en allant de gauche à droite. Il est généralement utile de commencer par écrire le type d'une nouvelle fonction; Dans ce cas, nous voyons que le type devrait être Arborescence a -> [a]. Ce qui revient à dire que frange est une fonction polymorphe qui, pour tout type a associe des arborescences de a à une liste de a. En voici une définition convenable :

```
fringe                     :: Tree a -> [a]
fringe (Leaf x)            =  [x]
fringe (Branch left right) =  fringe left ++ fringe right
```

Ici ++ est l'opérateur infixe qui concatène deux listes (sa définition complète sera donnée dans la section 9.1).

Comme dans l'exemple length donné plus tôt, la fonction frange est définie en utilisant une correspondance de motif, sauf qu'ici nous introduisons des motifs impliquant des constructeurs définis par l'utilisateur : Feuille et Branche. [Notons que les paramètres formels sont facilement identifiables par leurs noms qui commencent par une minuscule.]

II-3. Les synonymes de Type

Pour notre confort, Haskell permet de définir des synonymes de type; C'est à dire, des noms pour les types fréquemment utilisés. Les synonymes de type se créent en utilisant la déclaration type (4.2.2). En voici quelques exemples :

```
type String             = [Char]
type Person             = (Name,Address)
type Name               = String
data Address            = None | Addr String
```

Les synonymes de type ne définissent pas de nouveaux types, mais ils donnent simplement un nouveau nom à des types pré-existants. Par exemple, le type Personne -> Nom est strictement équivalent à (String,Adresse) -> String. Les nouveaux noms sont souvent plus courts que ceux dont ils sont synonymes, mais cela n'est pas l'unique avantage des synonymes de type : Ils peuvent également améliorer la lisibilité des programmes qu'ils rendent plus mnémotechniques. L'exemple précédent le démontre bien. Il est même possible de donner de nouveaux noms à des types polymorphes :

```
type AssocList a b              = [(a,b)]
```

Est le type de « listes associatives » qui associent des valeurs de type a à des valeurs de type b.

II-2-4. Les types internes n'ont rien de particulier

Précédemment, nous avons présenté plusieurs types « internes » tels que les listes, les tuples, les entiers et les caractères. Nous avons également montré comment de nouveaux

types pouvaient être définis par l'utilisateur. Hormis la syntaxe spécifique, les types internes ont-ils quoi que ce soit de particulier par rapport aux types définis par l'utilisateur ? La réponse est non. La syntaxe spécifique est une commodité et elle se conforme à des conventions historiques mais n'a aucune conséquence sémantique.

Pour mettre ce point en exergue, considérons la déclaration de type que nous écririons pour redéfinir ces types internes, si nous y étions autorisés, avec notre syntaxe spécifique. Par exemple, le type Char pourrait être ré-écrit de cette manière :

```
data Char         = 'a' | 'b' | 'c' | ...          -- This is not
valid
                | 'A' | 'B' | 'C' | ...         -- Haskell code!
                | '1' | '2' | '3' | ...
                ...
```

La syntaxe du nom de ces constructeurs n'est pas valide; Pour bien faire, il faudrait écrire quelque chose comme :

```
data Char        = Ca | Cb | Cc | ...
                 | CA | CB | CC | ...
                 | C1 | C2 | C3 | ...
```

Même si ces constructeurs sont plus concis, ce n'est pas une manière très conventionnelle de représenter des caractères.

Malgré tout, écrire un tel « pseudo-code » Haskell nous aide à mieux comprendre cette syntaxe spécifique. Cela montre que le type Char n'est qu'un type énumératif qui consiste en un grand nombre de constructeurs nulaires. Cette manière de représenter le type Char démontre que l'on peut utiliser une correspondance de motif sur des caractères dans la définition d'une fonction, comme nous pouvions nous attendre à pouvoir le faire pour n'importe quel constructeur de type.

[Cet exemple introduit l'usage des commentaires dans Haskell; Les caractères -- ainsi que tous ceux qui suivent jusqu'à la fin d'une ligne sont ignorés. Haskell permet également les commentaires emboîtés qui sont de la forme {-...-} et peuvent être insérés n'importe où (2.2).]

De la même manière, nous pourrions définir Int (ensemble fini d'entiers homogènes) et Integer avec :

```
data Int     = -65532 | ... | -1 | 0 | 1 | ... | 65532
                    -- more pseudo-code
data Integer =       ... -2 | -1 | 0 | 1 | 2 ...
```

ou -65532 et 65532 sont, respectivement, les valeurs minimum et maximum des entiers pour une implémentation donnée. Int est une énumération beaucoup plus grande que Char, mais elle est finie. En revanche, le pseudo-code pour Integer est conçu pour donner

une énumération infinie.

À ce petit jeu, les tuples sont également faciles à définir :

```
data (a,b)              = (a,b)        -- more pseudo-code
data (a,b,c)            = (a,b,c)
data (a,b,c,d)          = (a,b,c,d)
 .          .
 .          .
```

Chacune des déclaration ci-dessus définit un type tuple d'une longueur donnée, (...) ayant un rôle aussi bien dans la syntaxe de l'expression (en tant que constructeur de données) que dans la syntaxe de l'expression de type (en tant que constructeur de type). Les points verticaux après la dernière déclaration signalent un nombre infini de telles déclarations, pour indiquer qu'Haskell autorise les tuples de n'importe quelle longueur.

Les listes peuvent également être traitées facilement et, en prime, elles sont récursives :

```
data [a]               = [] | a : [a]    -- more pseudo-code
```

Nos précédentes explications concernant les listes s'expliquent plus clairement à présent : [] est la liste vide, et : est l'opérateur infixe de construction de liste. Par conséquent [1,2,3] doit être équivalent à la liste 1:2:3:[] (: est associatif à droite). Le type de [] est [a], et le type de : est a->[a]->[a].

[La syntaxe de notre pseudo-définition de « : » est légale. En effet, les constructeurs infixes sont autorisés dans les déclarations data, et on les distingue des opérateurs infixes (pour la correspondance de motifs) par le fait qu'ils doivent commencer par un « : » (exigence à laquelle « : » se conforme de manière rudimentaire.]

À ce stade le lecteur devrait soigneusement noter les différences entre les listes et les tuples; Les définitions que nous en avons faites ci-dessus mettent ces différences en évidence. Notons, en particulier, la nature récursive du type liste dont les éléments sont homogènes et de longueur arbitraire, ainsi que la nature non-récursive du type d'un tuple (particulier) dont les éléments sont hétérogènes et de longueur fixe. Les règles de typage pour les tuples et les listes devraient également être claires :

Pour (e1,e2,...,eN), n>=2, si ti est le type de ei, alors, le type du tuple est (t1,t2,...,tN).

Pour [e1,e2,...,eN], n>=0, chaque ei doit être du même type t, et le type de la liste est [t].

II-4-1. Les compréhensions de listes et les séquences arithmétiques

De même qu'avec les dialectes Lisp, les listes sont primordiales dans Haskell, et ainsi que dans d'autres langages fonctionnels, il existe beaucoup de friandises syntaxiques pour aider à leur création. En plus des constructeurs de listes abordés précédemment, Haskell

fournit une expression connue sous le nom de compréhension de liste qui s'explique mieux avec des exemples :

```
[ f x | x <- xs ]
```

Intuitivement, cette expression peut se lire « La liste de tout f x où x appartient à xs ». La ressemblance avec la notation des ensembles n'est pas une coïncidence. La phrase x <- xs est appelée un générateur, qui peuvent être plusieurs, comme dans :

```
[ (x,y) | x <- xs, y <- ys ]
```

Cette compréhension de liste forme le produit cartésien des deux listes xs et ys. Les éléments sont sélectionnés comme si les générateurs étaient « emboîtés » de gauche à droite (le générateur le plus à droite variant plus rapidement); Donc, si xs est [1,2] et ys est [3,4], le résultat est [(1,3),(1,4),(2,3),(2,4)].

Outre les générateurs, des expressions booléennes appelées gardes sont permises. Les gardes imposent des contraintes aux éléments générés. Par exemple, voici une définition concise de l'algorithme de tri que tout le monde aime :

```
quicksort  []           =  []
quicksort (x:xs)        =  quicksort [y | y <- xs, y<x ]
                        ++ [x]
                        ++ quicksort [y | y <- xs, y>=x]
```

Afin d'améliorer encore la manipulation des listes, Haskell offre une syntaxe spéciale pour les séries arithmétiques, que la série d'exemples suivante vous révèle :

```
[1..10]        =>   [1,2,3,4,5,6,7,8,9,10]
[1,3..10]      =>   [1,3,5,7,9]
[1,3..]        =>   [1,3,5,7,9, ... (infinite sequence)
```

Plus de détails seront donnés sur les séquences arithmétiques dans la section 8.2, et sur les « listes infinies » dans la section 3.4.

II-4-2. Chaînes de caractères

Un autre exemple de friandise syntaxique pour les types internes est révélé par le fait que la chaîne de caractères « bonjour » est en réalité un raccourci pour la liste de caractères ['b','o','n','j','o','u','r']. En effet, le type de « bonjour » est String, ou String est le synonyme d'un type prédéfini (que nous avons donné dans un exemple précédent) :

```
type String             = [Char]
```

Cela signifie que nous pouvons utiliser des fonctions polymorphes prédéfinies opérant sur

des listes pour traiter des chaînes de caractères. Par exemple :

"Bonjour" ++ "monde"   =>   "Bonjour monde"

III. Les Fonctions
III-1. Les abstractions lambda
III-2. Opérateurs infixes
III-2-1. Sections
III-2-2. Les déclarations de fixité
III-3. Les fonctions sont non-strictes
III-4. Les structures de données "infinies"
III-5. La fonction d'erreur

III. Les Fonctions

Étant donné que Haskell est un langage fonctionnel, on pourrait s'attendre à ce que les fonctions y jouent un rôle majeur...et on aurait raison. Dans cette section, nous abordons différents aspects des fonctions dans Haskell.

Pour commencer, considérons cette définition d'une fonction qui additionne ses deux arguments :

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

Ceci est un exemple de fonction curryfiée. Une application de add est de la forme add e1 e2, et est équivalente à (add e1) e2, puisque les applications des fonctions sont associatives à gauche. En d'autres termes, l'application de la fonction add au premier argument retourne une nouvelle fonction qui est à son tour "appliquée" au deuxième argument. Ceci est conforme au type de add, Integer->Integer->Integer qui est équivalent à Integer->(Integer->Integer). Puisque -> est associatif à droite. Nous pouvons redéfinir différemment la fonction inc (définie au chapitre précédent) en utilisant add :

```
inc = add 1
```

inc ainsi redéfinie est un exemple d'application partielle d'une fonction curryfiée, et c'est une façon, parmi d'autres, de retourner une fonction en tant que valeur. Considérons un cas ou le passage d'une fonction en tant qu'argument est utile. La fameuse fonction map en est un parfait exemple :

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

[Les applications de fonctions ont la priorité sur n'importe quel opérateur infixe et, par conséquent, le terme droit de la seconde équation est équivalent à (f x) : (map f xs)] La fonction map est polymorphe et son type indique clairement que son premier argument est une fonction ; Notons également que les occurrences des deux a doivent être de même type (il en est de même pour les b). Pour voir map en action, nous pouvons incrémenter les éléments d'une liste :

```
map (add 1) [1,2,3] => [2,3,4]
```

Ces exemples démontrent que les fonctions sont bien de « première-classe » et elles sont souvent appelées fonctions d'ordre supérieur lorsqu'elles sont utilisées ainsi.

III-1. Les abstractions lambda

Plutôt que d'utiliser des équations pour définir les fonctions, nous pouvons aussi les définir « anonymement » via une abstraction lambda. On pourrait, par exemple, définir une fonction équivalente à \x -> x+1. De la même manière, la fonction add est équivalente à \x -> \y -> x+y. Les abstractions lambda emboîtées telle que celle-ci peuvent être écrites en utilisant la notation raccourcies équivalente \x y -> x+y. En réalité, les équations :

```
inc x                  = x+1
add x y                = x+y
```

ne sont que des raccourcis pour :

```
inc                    = \x    -> x+1
add                    = \x y  -> x+y
```

Nous aurons encore à dire sur de telles équivalences plus tard. En général, sachant que x est de type t1 et exp est de type t2, alors \x->exp est de type t1 -> t2.

III-2. Opérateurs infixes

Les opérateurs infixes ne sont que des fonctions, et peuvent aussi être définis à l'aide d'équations. Voici, par exemple, la définition d'un opérateur de concaténation de listes.

```
(++)                    :: [a] -> [a] -> [a]
[]      ++ ys           =  ys
(x :xs) ++ ys           =  x   : (xs++ys)
```

[D'un point de vue lexical, les opérateurs doivent être des « symboles », contrairement aux identificateurs normaux qui sont alpha-numériques (§2.4). Dans Haskell, il n'y a pas d'opérateurs préfixes, à l'exception du signe moins (-) qui peut aussi bien être infixé que préfixé.]

Un autre exemple est celui du précieux opérateur infixe pour la composition de fonctions :

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

III-2-1. Sections

Puisque les opérateurs infixes ne sont que des fonctions, il est logique qu'ils puissent aussi

être utilisés dans des applications partielles. Dans Haskell, une appllication partielle sur un opérateur infixe est appelée une section. Par exemple :

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

[Les parenthèses sont obligatoires.]

La dernière forme de section donnée ci-dessus astreint un opérateur infixe à une valeur de fonction équivalente, ce qui est pratique lorsque l'on doit passer un opérateur infixe en tant qu'argument à une fonction, par exemple dans map (+) [1,2,3] (le lecteur devrait s'assurer, par lui même, que ceci retourne bien une liste de fonctions !). Cela est également nécessaire lorsque l'on définit la signature de type d'une fonction, comme dans les exemples de (++) et de (.) donnés précédemment.

Nous pouvons maintenant voir que add défini précédemment n'est que (+), et inc n'est que (+1) ! En fait, les définitions suivantes feraient aussi bien l'affaire :

```
inc                    = (+ 1)
add                    = (+)
```

Nous pouvons astreindre des opérateurs infixes à prendre une valeur de fonction équivalente, mais l'opération inverse est-elle possible ? Oui ! Il suffit de mettre l'identificateur lié à une valeur de fonction entre guillemets inversés. Par exemple, x 'add' y est l'équivalent de add x y.

Certaines fonctions se comprennent mieux de cette manière. Un bon exemple en est le prédicat prédéfini d'appartenance à une liste elem ; L'expression x 'elem' xs peut intuitivement se lire "x est un élément appartenant à xs".

[Il existe quelques règles particulières concernant les sections incluant l'opérateur préfixe/infixe -]

À ce stade, le lecteur pourrait être un peu perdu par les innombrables façons de définir une fonction ! La décision de fournir ces mécanismes est due en partie à des conventions historiques, et reflète en partie un désir de cohérence (par exemple, dans le traitement accordé aux infixes "contre" les fonctions).

III-2-2. Les déclarations de fixité

Une déclaration de fixité peut être déclarée pour n'importe quel opérateur infixe ou constructeur (y compris ceux fait d'identificateurs ordinaires, tel que 'elem'). Cette déclaration permet de spécifier un niveau de priorité allant de 0 à 9 (9 étant le plus élevé ; Le niveau de priorité supposé d'une application normale étant 10), ainsi que left- (gauche), right- (droite), ou non-associativité. Par exemple, les déclarations de fixité de ++ et . sont :

```
infixr 5 ++
infixr 9 .
```

Ces deux exemples spécifient une associativité à droite, la première avec un niveau de priorité 5 et la dernière un niveau 9. L'associativité à gauche se spécifie via infixl, et la non-associativité via infix. En outre, il est possible de spécifier la fixité de plusieurs opérateurs dans la même déclaration. Si aucune déclaration de fixité n'est donnée pour un opérateur particulier, la fixité par défaut infixl 9 sera attribuée. (Voir §5.9 pour une définition détaillée des règles d'associativité).

III-3. Les fonctions sont non-strictes

Supposons que bot est définie par :

```
bot                          = bot
```

En d'autres termes, bot est une expression sans fin ou non-finie. La valeur de telles expressions est abstraitement dénotée par la valeur ⊥ (lire « bottom » qui signifie « fond » en anglais). Cette valeur est également attribuée aux expressions dont le résultat aboutit à une erreur d'exécution, telle que 1/0. Une telle erreur est irrécupérable : Les programmes ne peuvent pas continuer après ces erreurs. Les erreurs rencontrées par le système d'entrées/sorties, telle qu'une erreur de fin-de-fichier, sont récupérables et sont traitées différemment. (De telles erreurs d'entrées/sorties ne sont pas réellement des erreurs, mais plutôt des exceptions. Les exceptions seront traitées en détail dans la section 7).

Une fonction f est dite stricte si, lorsqu'elle est appliquée à une expression non-finie, elle ne se termine pas non plus. En d'autres termes, f est stricte ssi (si et seulement si) la valeur de f bot est ⊥. Dans la plupart des langages de programmation, toutes les fonctions sont strictes. Mais ce n'est pas le cas dans Haskell. Considérons, pour l'exemple, const1, la fonction constante 1, définie par :

```
const1 x                     = 1
```

Dans Haskell, la valeur de const1 bot est 1. D'un point de vue opérationnel, puisque const1 n'a pas "besoin" de la valeur de son argument, celui-ci n'est jamais évalué et, par conséquent, const1 ne provoque jamais de calcul sans fin. Pour cette raison, les fonctions non-strictes sont également appelées des "fonctions paresseuses", et on dit qu'elles évaluent leurs arguments avec "paresse", ou "à la demande" (en anglais on parle de "lazy evaluation").

Le paragraphe ci-dessus vaut aussi pour les erreurs, puisque les erreurs et les valeurs non-finies sont sémantiquement identiques pour Haskell. Par exemple, l'évaluation de const1 (1/0) retourne correctement 1.

Les fonctions non-strictes sont extrêmement utiles dans plusieurs contextes. L'avantage principal est qu'elles libèrent le programmeur de beaucoup de soucis concernant l'ordre des évaluations. Des valeurs qui exigent un calcul lourd peuvent être passées à des fonctions sans craindre qu'elles soient calculées si cela n'est pas nécessaire. Un exemple significatif est celui d'une structure de données qui peut être infinie.

Abordons les fonctions non-strictes sous un autre angle : Les calculs opérés par Haskell se basent sur des "définitions" plutôt que sur les "assignations" utilisées dans les langages traditionnels. Une déclaration telle que :

```
v                        = 1/0
```

se lit : "définir v comme étant 1/0" mais ne se lit pas : "calculer 1/0 et stocker le résultat dans v". C'est uniquement quand la valeur (définition) de v sera nécessaire que la division par zéro se produira. Par elle même, cette déclaration n'implique aucun calcul. La programmation par assignations requiert une grande attention dans l'ordre des assignations : La signification d'un programme dépend de l'ordre dans lequel ces assignations sont exécutées. À l'inverse, les définitions sont beaucoup plus simples : Elles peuvent être présentées dans n'importe quel ordre sans affecter la signification d'un programme.

III-4. Les structures de données "infinies"

Un des avantages de la nature non-stricte de Haskell est que les constructeurs de données sont également non-stricts. Ce n'est pas surprenant, puisque les constructeurs ne sont jamais que des fonctions d'un certain type (ils ne se distinguent des fonctions que parce qu'ils peuvent être utilisés pour une correspondance de motif). Par exemple, le constructeur de listes, (:), est non-strict.

Les constructeurs non-stricts permettent la définition de structures de données (conceptuellement) infinies. Voici une liste infinie de uns :

```
uns                     = 1  : uns
```

La fonction numsFrom est peut-être encore plus intéressante :

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

Donc numsFrom n est la liste infinie des entiers successifs à partir de n. Nous pouvons créer un liste infinie de carrés à partir de numsFrom :

```
squares                  = map (^2) (numsFrom 0)
```

(Notez l'utilisation d'une section ; ^ étant l'opérateur infixe d'exponentiation.)

Bien entendu, nous souhaitons éventuellement extraire une portion finie de cette liste pour un calcul et il existe un grand nombre de fonctions dans Haskell qui font ce genre de choses : take, takewhile, filter pour ne citer qu'elles. Les définitions de Haskell incluent un large ensemble de fonctions et de types internes — Cela se nomme le "Prélude Standard". Le Prélude Standard complet est inclus dans la section 8 de la première partie du Haskell report ; Voir la partie nommée PreludeList pour les fonctions opérant sur des listes. Par exemple, take retourne les n premiers éléments d'une liste :

```
take 5 squares => [0,1,4,9,16]
```

La définition de uns ci-dessus est un exemple de liste circulaire. Dans la plupart des circonstances, la paresse a un impact important sur l'efficacité, puisqu'on peut attendre d'une implémentation qu'elle implémente cette liste comme une réelle structure circulaire, économisant ainsi la mémoire.

Un autre exemple de l'utilisation de la circularité est la suite de Fibonacci qui peut être calculée efficacement avec la suite infinie suivante :

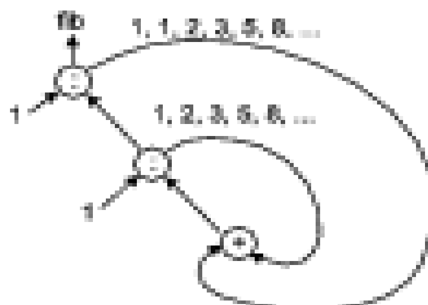
```
fib             = 1  : 1  : [ a+b | (a,b) <- zip fib (tail fib) ]
```

ou zip est une fonction du Prélude Standard qui retourne une liste de paires composées des éléments de même position dans les deux listes.

```
zip (x :xs) (y :ys)     = (x,y)  : zip xs ys
zip  xs     ys          = []
```

Remarquez de quelle manière fib, une liste infinie, est définie en fonction d'elle même, comme si elle courrait après sa propre queue. Un schéma de ce calcule figure sous 1.

Pour une autre définition de liste infinie, voir section 4.4



Suite de Fibonacci circulaire

III-5. La fonction d'erreur

Haskell a une fonction interne appelée error dont le type est String->a. C'est une fonction un peu étrange : à en juger par son type, il semble qu'elle retourne la valeur d'un type polymorphe à propos duquel elle ne sait rien, puisqu'à aucun moment un argument de ce type ne lui est passé.

En fait, il y a une valeur "partagée" par tous les types : ⊥. En effet, c'est cette valeur qui est

systématiquement retournée par une erreur (rappelez-vous que les erreurs ont la valeur $\perp$). Cependant, nous pouvons attendre d'une implémentation raisonnable qu'elle envoie l'argument String à error à des fins de diagnostic. Cette fonction est donc utile lorsque nous voulons qu'un programme se termine si quelque chose a "mal tourné". Par exemple, la définition réelle de head du Prélude Standard est :

```
head (x :xs)            =  x
head  []                =  error "head{PreludeList} : head []"
```

IV. Les Expressions Case et la correspondance de motifs
IV-1. La sémantique des correspondances de motifs
IV-2. Un exemple
IV-3. Les expressions casuelles (Case expressions)
IV-4. Les motifs paresseux (Lazy-patterns)
IV-5. Cadrage lexical et formes emboîtées
IV-6. Mise en forme

IV. Les Expressions Case et la correspondance de motifs

Prédécemment, nous avons donné plusieurs exemples de correspondance de motifs dans la définition de fonctions, par exemple length et frange. Dans cette section, nous aborderons le processus des correspondances de motifs de manière plus détaillée (§3.17). Dans Haskell, la correspondance de motifs est différente de celle que l'on trouve dans les langages de programmation logique tel que Prolog. En particulier, on peut la voir comme une correspondance à sens unique, alors que Prolog permet une correspondance bi-directionnelle (via l'unification), avec la rétro-recherche implicite de son méchanisme d'évaluation.

Les correspondances ne sont pas de « première-classe ». Il n'existe qu'un ensemble limité de correspondances différentes. Nous avons déjà vu plusieurs exemples de correspondances de motifs dans les constructeurs de données. Les fonctions length et frange définies précédemment utilisent toutes deux ce type de correspondance, la première sur le constructeur d'un type «pré-défini» (les listes), et la deuxième sur un type défini par l'utilisateur (Arborescence). En effet, la correspondance de motifs est autorisée sur des constructeurs de n'importe quel type, qu'il soit défini par l'utilisateur ou pas. Y inclut les tuples, chaînes de caractères, nombres, caractères, etc. Par exemple, voci une fonction contrainte qui utilise une correspondance de motifs sur un tuple de « constantes » :

```
contrainte :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrainte    ([],  'b',  (1,   2.0),   "hi",   True) = False
```

Cet exemple démontre qu'il est possible d'emboîter des motifs (sans limite de profondeur).

Techniquement parlant, les paramètres formels que le Haskell Report les nomme variables sont également des motifs. La seule différence est que la correspondance avec une valeur n'échoue jamais. L'« effet secondaire » (ou « effet de bord ») d'une correspondance positive est que le paramètre formel est lié à la valeur du motif utilisé dans la correspondance. C'est pour cette raison que, dans toute équation, il ne peut y avoir plus d'une occurence du même paramètre formel (une propriété que l'on nomme linéarité §3.17, §3.3, §4.4.2).

Les motifs tels que les paramètres formels dont la correspondance n'échoue jamais sont dits irréfutables, contrairement aux motifs réfutables dont la correspondance peut

échouer. Le motif utilisé dans l'exemple contrainte ci-dessus est réfutable. Il existe trois autres types de motifs irréfutables, et nous présentons deux d'entre eux maintenant (le dernier ne sera abordé que dans la section 4.4).

Les motifs nommés (As-patterns)

Il est parfois pratique de nommer un motif pour l'utiliser dans le terme droit d'une équation. Par exemple, une fonction qui duplique le premier élément dans une liste peut être définie avec :

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

(Rappelez-vous que « : » est associatif à droite.) Notez que x :xs apparaît à la fois comme motif dans le terme gauche et comme expression dans le terme droit. Pour améliorer la lisibilité, il pourrait être préférable de n'écrire x :xs qu'une seule fois, ce qui peut être réalisé en utilisant un motif nommé (as-pattern) comme suit : un autre avantage de cette nouvelle définition est qu'elle empêche une implémentation maladroite qui aboutirait à la reconstruction complète de x :xs plutôt que de réutiliser la valeur sur laquelle la correspondance est opérée.

```
f s@(x :xs)               = x :s
```

Techniquement parlant, les correspondances sur des motifs nommés sont toujours positives, même si la correspondance sur le sous-motif (dans ce cas x :xs) peut, bien sûr, échouer.

Les joker (wild-cards)

Une autre situation rencontrée fréquemment est la correspondance de motifs avec une valeur, qui n'est pas importante. Par exemple les fonctions head et tail définies dans la section 2.1 peuvent être redéfinies avec :

```
head (x :_)               = x
tail (_ :xs)              = xs
```

dans ce cas nous avons « fait savoir » que certaines parties du paramètre en entrée sont sans intérêt pour nous. Chaque joker correspond indépendamment à n'importe quelle valeur, mais contrairement à un paramètre formel, aucune valeur n'est liée. Pour cette raison, il peut y en avoir plus d'un dans une équation.

IV-1. La sémantique des correspondances de motifs

À ce stade nous avons montré comment la correspondance était réalisée sur des motifs individuels, que certains étaient réfutables et d'autres irréfutables, etc. Mais qu'est-ce qui conduit ce processus dans son ensemble? Dans quel ordre les correspondances sont-elles

testées? Qu'arrive-t-il si toutes échouent? Cette section aborde ces questions.

La correspondance de motifs peut soit échouer soit réussir ou encore diverger. Une correspondance positive (ou réussie) lie les paramètres formels dans le motif. Une divergence intervient lorsqu'une valeur requise dans le motif contient une erreur (⊥). Le processus de mise en correspondance se fait de haut en bas et de gauche à droite. Si une correspondance échoue ou que ce soit dans une équation, toute l'équation échoue et l'équation suivante est testée. Si toutes les équations échouent, la valeur de l'application de la fonction est ⊥, et une erreur à l'exécution en résulte.

Par exemple, si [1,2] est comparé à [0,bot], alors 1 ne correspond pas à 0 et la correspondance échoue. (Rappelez vous que bot, défini prédécemment, est une variable liée à ⊥). Mais si [1,2] est comparé à [bot,0], alors la comparaison 1 à bot provoque une divergence (c.-à-d. ⊥).

L'autre particularité de cet ensemble de règles est que les motifs de niveau supérieur peuvent aussi être accompagnés de gardes booléens, comme dans cette définition d'une fonction qui retourne le signe d'un nombre :

```
sign x |   x >  0          =   1
       |   x == 0          =   0
       |   x <  0          =  -1
```

Notez qu'une séquence de gardes peut être fournie pour le même motif. Comme pour les motifs, ils sont évalués de haut en bas et de gauche à droite, et le premier dont l'évaluation retourne True (vrai) aboutit à une correspondance positive.

IV-2. Un exemple

Les règles de correspondance de motifs peuvent avoir de subtils effets sur la signification d'une fonction. Par exemple, considérons cette définition de take :

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

et cette version légèrement différente (les deux premières équations ont été interverties) :

```
take1  _      []          =  []
take1  0      _           =  []
take1  n     (x:xs)       =  x : take1 (n-1) xs
```

Maintenant notez ce qui suit :

```
take  0   bot  => []
take1 0   bot  => ⊥
take  bot []   => ⊥
take1 bot []   => []
```

nous voyons que take est « plus définie » concernant son deuxième argument, alors que take1 est plus définie concernant sont premier argument. Il est difficile de dire, dans ce cas, quelle définition est la meilleure. Rappelez-vous simplement que dans certaines applications, cela peut faire une différence. (Le Prélude Standard inclut une définition correspondante à take).

IV-3. Les expressions casuelles (Case expressions)

La correspondance de motifs fournit un moyen de répartir des contrôles basés sur les propriétés structurelles d'une valeur. Dans bien des cas, nous ne souhaitons pas définir une fonction chaque fois qu'une correspondance de motifs est nécessaire, mais à ce stade nous avons uniquement abordé les correspondances de motifs dans la définition des fonctions. Les expressions casuelles dans Haskell permettent de résoudre ce problème. En fait, dans le Haskell Report, la signification des correspondances de motifs dans les définitions de fonctions est spécifiée en termes d'expressions casuelles, qui sont considérées comme étant plus primitives. En particulier, la définition d'une fonction de la forme :

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

ou chaque pij est un motif, est sémantiquement équivalente à :

```
f x1 x2 ... xk = case (x1, ... , xk) of
 (p11, ... ,p1k) -> e1
...
 (pn1, ... ,pnk) -> en
```

ou les xi sont de nouveaux identificateurs (pour une explication plus générale incluant les gardes, voir §4.4.2). Par exemple, la définition de take donnée plus haut est équivalente à :

```
take m ys                = case (m,ys) of
                               (0,_)       ->  []
                               (_,[])      ->  []
                               (n,x :xs)   ->  x : take (n-1) xs
```

Pour être complet à propos des types, un point que nous n'avons pas encore précisé : les types dans le terme droit d'une expression casuelle ou d'un ensemble d'équations comprenant une définition de fonction, doivent tous être les mêmes. Plus précisément, ils doivent tous partager le même type principal commun.

Les règles de correspondance de motifs pour les expressions casuelles sont les mêmes que celles données pour les définitions de fonctions, il n'y a donc rien de nouveau à apprendre ici, si ce n'est de noter le confort offert pas les expressions casuelles. En fait, il y a un usage des expressions casuelles qui est si courant qu'il a sa propre syntaxe : L'« expression

conditionnelle ». Dans Haskell, les expressions conditionnelles ont la forme familière :

```
if e1 then e2 else e3
```

qui n'est jamais qu'un raccourci pour :

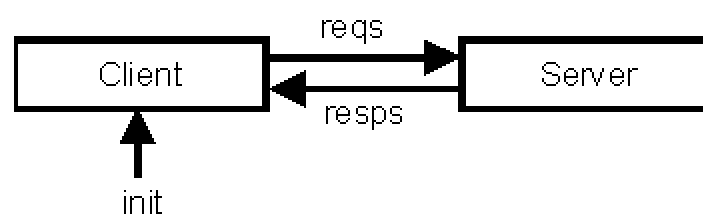
```
case e1 of True  -> e2
           False -> e3
```

À partir de là il devrait être clair que e1 doit être de type Bool, et que e2 et e3 doivent être de même type (arbitrairement défini). En d'autres termes, if-then-else vu en tant que fonction est de type Bool->a->a->a.

IV-4. Les motifs paresseux (Lazy-patterns)

Il existe un autre type de motif autorisé dans Haskell. Il est nommé motif paresseux, et est de la forme ~pat. Les motifs paresseux sont irréfutables : une correspondance de la valeur v avec ~pat est toujours positive, quel que soit pat. D'un point de vue opérationnel, si un identificateur dans pat devait être « utilisé » plus tard dans le terme droit, il sera lié à la portion de la valeur qui résulterait de la correspondance entre pat et v, et ⊥ autrement.

Les motifs paresseux sont utiles dans les contextes ou des structures de données infinies sont définies récursivement. Par exemple, les listes infinies sont un excellent vecteur pour écrire des programmes de simulation, et dans ce contexte les listes infinies sont souvent appelées flux. Considérons le cas simple de la simulation d'une interaction entre un processus serveur server et un processus client client, où client envoie une séquence de requêtes au server, et server répond à chaque requête par une réponse. Cette situation est schématisée dans la figure suivante. (Notez que client accepte un message initial en tant qu'argument).



Simulation client-serveur

En utilisant des flux pour simuler la séquence de messages, le code Haskell correspondant à ce schéma est :

```
reqs                         = client init resps
resps                        = server reqs
```

Ces équations récursives sont une traduction lexicale directe du schéma.

Supposons que la structure du serveur et du client ressemblent à cela :

```
client init (resp :resps) = init : client (next resp) resps
server      (req :reqs)   = process req : server reqs
```

où nous supposons que next est une fonction qui, une réponse du serveur ayant été reçue, détermine la requête suivante, et process est une fonction qui traite une requête du client, retournant une réponse appropriée.

Malheureusement, ce programme a un sérieux problème : Il ne produira aucune sortie! Le problème est que le client, tel qu'utilisé dans l'appel récursif de reqs et resps, teste une correspondance sur la liste de réponses avant d'avoir soumis sa première requête! En d'autres termes, la correspondance de motif intervient « trot tôt ». Une manière de résoudre ce problème est de redéfinir client comme suit :

```
client init resps = init : client (next (head resps)) (tail resps)
```

Même si elle fonctionne, cette solution ne se lit pas aussi aisément que la définition précédente. Une meilleure approche consiste à utiliser un motif paresseux :

```
client init ~(resp :resps) = init : client (next resp) resps
```

Puisque les motifs paresseux sont irréfutables, la correspondance est immédiatement positive, permettant à la requête initiale d'être « soumise », ce qui, en retour, permet à la première réponse d'être générée. Le moteur est donc « lancé », et la récursion s'occupe du reste.

Pour une démonstration de ce programme en action, si nous définissons :

```
init                   = 0
next resp              = resp
process req            = req+1
```

alors nous voyons que :

```
take 10 reqs => [0,1,2,3,4,5,6,7,8,9]
```

Pour un autre exemple de l'utilisation des motifs paresseux, considérons la définition de Fibonacci donnée précédemment :

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

Nous pourrions essayer de la ré-écrire en utilisant un motif nommé :

```
fib@(1 :tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

Cette version de fib a le (léger) avantage de ne pas utiliser tail dans le terme droit, puisque qu'il est disponible dans une forme « destructurée » dans le terme gauche en tant que tfib.

[Ce genre d'équation est appelée un motif de liaison (pattern binding) parce que c'est une équation niveau supérieur dans laquelle l'intégralité du terme gauche est un motif, c.-à-d. fib et tfib sont des liens à l'intérieur du cadre de la déclaration.]

En reprenant le même raisonnement que tout à l'heure, nous serions tentés de croire que ce programme ne générera aucune sortie. Et pourtant, il le fait, et la raison en est simple : Dans Haskell, les motifs de liaison sont supposés avoir un ~ implicite à leur gauche, reflétant le comportement le plus commun attendu d'un motif de liaison, et évitant des situations anormales qui dépassent le cadre de ce tutoriel. Nous voyons donc que les motifs paresseux jouent un rôle important dans Haskell, ne serait-ce qu'implicitement.

IV-5. Cadrage lexical et formes emboîtées

Il est souvent souhaitable de créer un cadre emboîté à l'intérieur d'une expression, dans le but de créer des liens locaux qui ne seront pas visibles ailleurs—c.-à-d. une espèce de forme structurée en blocs. Dans Haskell, il y a deux manière de réaliser cela :

Les expressions Let

Les expressions Let dans Haskell sont utiles chaque fois qu'un lien emboîté est requis. Pour en donner un exemple simple, considérons :

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

L'ensemble de liaisons créé par une expression let est mutuellement récursif , et les motifs de liaison sont traités comme des motifs parresseux (c.-à-d. qu'ils sont implicitement précédés d'un ~). Les seuls types de déclaration autorisées sont la signature de type, les liaisons de fonctions, et les liaisons de motifs.

Les clauses where

Il est parfois pratique de délimiter les liaisons de plusieurs équations incluant une garde, ce qui nécessite une clause where :

```
f x y  |  y>z           =  ...
       |  y==z          =  ...
       |  y<z           =  ...
     where z = x*x
```

Notez que l'on ne peut pas faire cela avec une expression let, qui délimite uniquement l'expression qu'elle renferme. Une clause where n'est autorisée qu'au niveau supérieur d'un ensemble d'équations ou d'une expression casuelle. Les propriétés et contraintes sur les liaisons dans une expression let s'appliquent également à celles dans les clauses where.

Ces deux formes de cadres emboîtés sont très proches, mais rappelez-vous qu'une expression let est une expression, alors qu'une clause where n'en est pas une—elle fait partie de la syntaxe d'une déclaration de fonction ou d'une expression casuelle.

IV-6. Mise en forme

Le lecteur se demande peut-être : comment Haskell évite l'utilisation des point-virgules, ou tout autre délimiteur, pour marquer la fin des équations, déclarations, etc. Par exemple, considérons cette expression let de la section précédente :

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

Comment l'analyseur syntaxique s'y prend-il pour ne pas interpréter cela comme l'équivalent de :

```
let y   = a*b f
    x   = (x+y)/y
in f c + f d
```

?

La réponse est que Haskell utilise une syntaxe à deux dimensions appelée mise en forme (ou layout) qui repose essentiellement sur le fait que les déclarations sont « alignées en colonnes ». Dans l'exemple ci-dessus, notez que y et f commencent à la même colonne. Les règles qui s'appliquent à cette mise en forme sont données en détail dans le Haskell Report (voir §2.7, §B.3), mais en pratique, cette mise en forme est plutôt intuitive. Il faut simplement se rappeler deux choses :

Premièrement, le caractère qui détermine la colonne de départ pour les déclarations dans les clauses where, let, ou les expressions casuelles est celui qui suit immédiatement les mots-clefs where, let ou of (cette règle s'applique également à where quand il est utilisé dans une déclaration de classe ou d'instance qui seront introduites dans la section 5). Par conséquent, nous pouvons démarrer la déclaration sur la même ligne que le mot-clef, la ligne suivante, etc. (Le mot-clef do, qui sera abordé plus tard, utilise aussi cette mise en forme).

Deuxièmement, il faut s'assurer que la colonne de départ est décalée à droite par rapport à la colonne de départ associée à la clause qui l'englobe (pour éviter tout ambiguïté). La « fin » d'une déclaration intervient lorsque quelque chose apparait à la colonne associée avec cette forme de liaison, ou à gauche de celle-ci. Haskell respecte la convention qui veut qu'une tabulation vaille 8 espaces. Il faut donc faire attention avec les éditeurs qui peuvent respecter d'autres conventions.

La mise en forme est en réalité un raccourcis pour un méchanisme de groupage explicite,

qui mérite d'être mentionné puisqu'il peut être utile dans certaines circonstances. L'exemple let donné plus haut est équivalent à :

```
let { y   = a*b
      ; f x = (x+y)/y
    }
in f c + f d
```

Notez les crochets cursifs et le point-virgule explicites. Un cas ou cette notation explicite est utile est lorsque l'on souhaite placer plus d'une déclaration sur une ligne. Par exemple, voici une expression valide :

```
let y   = a*b;  z = a/b
    f x = (x+y)/z
in f c + f d
```

Pour un autre exemple de délimitation explicite, voir §2.7. L'utilisation de la mise en forme réduit la confusion associée aux listes de déclarations, et améliore donc la lisibilité. Elle est facile à apprendre et son utilisation est encouragée.

V. Les classes de types et la surcharge

V. Les classes de types et la surcharge

Il y a une dernière particularité du système de typage d'Haskell qui le sépare des autres langages de programmation. Le genre de polymorphisme dont nous avons parlé jusque là est communément appelé polymorphisme paramétrique. Il existe un autre genre appelé polymorphisme ad hoc, plus connu sous le nom de surcharge. Voici quelques exemples de polymorphisme ad hoc :

* Les littéraux 1, 2, etc. sont souvent utilisés pour représenter à la fois les entiers à longueur fixe ou à longueur arbitraire.
* Les opérateurs numériques tel que + sont souvent définis pour opérer sur plusieurs types de nombres.
* L'opérateur d'égalité (== dans Haskell) opère sur des nombres mais également sur beaucoup (mais pas tous) d'autres types.

Notez que le comportement de ces opérateurs surchargés est différent pour chaque type (en fait, le comportement est parfois indéfini ou une erreur), tandis qu'avec le polymorphisme paramétrique le type n'a pas d'influence (frange, par exemple, ne se soucie pas de savoir quel type d'éléments se trouve dans les feuilles de son arborescence). Dans Haskell, les classes de types permettent de contrôler de manière structurée les polymorphismes ad hoc, ou surcharge.

Commençons avec un exemple simple mais important : l'égalité. Il y a beaucoup de types pour lesquels nous voudrions que l'égalité soit définie, mais d'autres pour lesquels nous ne le voulons pas. Par exemple, on considère généralement qu'il est trop difficle d'informatiser le calcul de l'égalité entre des fonctions, par contre nous avons souvent besoin de comparer l'égalité de deux listes. Le genre d'égalité dont nous parlons ici est l'« égalité de valeurs », à l'opposé de l'« égalité de pointeurs » que l'on trouve, par exemple, avec le == dans java. L'égalité de pointeurs n'est pas référentiellement transparente et ne s'adapte pas bien dans un langage purement fonctionnel. Pour mettre en évidence ce sujet, considérons cette définition d'une fonction elem qui test l'appartenance à une liste :

```
x ‘elem‘  []              = False
x ‘elem‘ (y :ys)          = x==y || (x ‘elem‘ ys)
```

[Pour des raisons stylistiques disutés dans la section 3.1, nous avons choisi de définir elem dans une forme infixée. == et || sont, respectivement, les opérateurs infixes de l'égalité et le ou logique.]

Intuitivement, the type de elem « doit » être : a->[a]->Bool. Mais cela impliquerait que == est de type a->a->Bool, quand bien même nous venons de dire que nous ne nous attendions pas à ce que == soit défini pour tous les types.

De plus, comme nous l'avons noté plus haut, même si == était défini pour tous les types, la comparaison de deux listes pour tester leur égalité ou la comparaison de deux entiers sont deux choses très différentes. En ce sens, nous nous attendons à ce que == soit surchargé pour mener ces différentes tâches.

Les classes de types permettent de résoudre ces deux problèmes. Elles permettent de déclarer tels types comme des instances de telles classes, et de fournir des définitions pour les opérations surchargées associées à une classe. Par exemple, définissons une classe de type contenant un opérateur d'égalité :

```
class Eq a where
  (==)                  :: a -> a -> Bool
```

Ici, Eq est le nom de la classe que l'on défini, et == est l'unique opérateur dans cette classe. Cette déclaration peut être lue « un type a est une instance de la classe Eq si il existe une opération (surchargée) ==, de type approprié, défini pour celui-ci ». (Notez que == est uniquement défini pour des paires d'objets de même type.)

La contrainte qui impose que a doit être une instance de la classe Eq s'écrit Eq a. Donc Eq a n'est pas une expression de type, mais plutôt l'expression d'une contrainte sur un type, que l'on appelle un contexte. Les contextes sont placés au début d'expressions de type. Par exemple, l'effet de la déclaration de classe ci-dessus est d'assigner le type suivant à == :

```
(==)                    :: (Eq a) => a -> a -> Bool
```

Ceci devrait se lire « Pour tout type a étant une instance de la classe Eq, == est de type a->a->Bool ». C'est ce type qui serait utilisé pour == dans l'exemple elem, et en fait, la contrainte imposée par le contexte se propage au type principal de elem :

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

Ce qui se lit « Pour tout type a étant une instance de la classe Eq, elem est de type a->[a]->Bool ». C'est exactement ce que nous voulons—cela exprime le fait que elem n'est pas défini pour tous les types, mais seulement pour ceux dont nous savons déterminer l'égalité des éléments.

Jusque là, tout va bien. Mais comment faire pour spécifier quels types sont des instances de la classe Eq, ainsi que le comportement de == sur chacun de ces types? Cela ce fait à l'aide d'une déclaration d'instance. Par exemple :

```
instance Eq Integer where
  x == y                =  x `integerEq` y
```

Cette définition de == est appelée une méthode. La fonction integerEq est une fonction primitive qui teste l'égalité entre deux entiers, mais en général n'importe quelle expression valide est autorisée dans le terme droit, comme pour n'importe quelle définition de

fonction. Dans son ensemble, cette déclaration signifie « Le type Integer est une instance de la classe Eq, et voici la définition de la méthode correspondant à l'opération == ». Cette déclaration donnée, nous pouvons maintenant tester l'égalité de l'ensemble fini d'entiers en utilisant ==. De la même manière :

```
instance Eq Float where
  x == y                =  x `floatEq` y
```

nous permet de comparer des nombres à virgule flottante en utilisant ==. Les types récursifs tel que Arborescence définie précédemment peuvent aussi être gérés :

```
instance (Eq a) => Eq (Arborescence a) where
  Feuille a         ==  Feuille b          =  a == b
  (Branche l1 r1)   ==  (Branche l2 r2)    =  (l1==l2) && (r1==r2)
  _                 ==  _                  =  False
```

Notez le contexte Eq a sur la première ligne—il est nécessaire parce que les éléments dans les feuilles (de type a) sont comparés pour tester leur égalité sur la seconde ligne. Cette contrainte additionnelle dit essentiellement que nous pouvons comparer des arborescences de a pour tester leur égalité à condition que nous soyons capables de tester l'égalité des a. Si le contexte était omis dans la déclaration d'instance, une erreur de type statique en résulterait.

Le Haskell Report, et particulièrement le Prélude, contient nombre d'exemples utiles de classes de types. En fait, une classe Eq y est définie et est légèrement plus longue que celle que nous avons définie plus haut :

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

C'est un exemple de classe comprenant deux opérations, une pour l'égalité, l'autre pour l'inégalité. C'est aussi une démonstration de l'utilisation d'une méthode par défaut, dans ce cas pour l'opération d'inégalité /=. Si la méthode d'une opération particulière est omise dans une déclaration d'instance, alors la méthode par défaut définie dans la déclaration de classe, si elle existe, est utilisée. Par exemple, les trois instances de Eq définies plus tôt fonctionneront parfaitement avec la déclaration de classe ci-dessus, fournissant exactement la définition de l'inégalité que nous voulons : la négation logique de l'égalité.

Haskell supporte également la notion d'extension de classe. Par exemple, nous pourions avoir besoin de définir une classe Ord qui hérite de toutes les opérations définies dans Eq, mais qui disposerait, en plus, d'un ensemble d'opérations de comparaison et des fonctions minimum et maximum :

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

Notez le contexte dans la déclaration de class. Nous disons que Eq est une superclasse de Ord (à l'inverse, Ord est une sous-classe de Eq), et tout type étant une instance de Ord doit aussi être une instance de Eq. (Dans la section suivante nous donnons une définition plus complète de Ord extraite du Prélude)

Un des bénéfices de telles inclusions de classes est que les contextes sont plus courts : Une expression de type pour une fonction qui utilise à la fois Eq et Ord peut utiliser le contexte (Ord a), plutôt que (Eq a, Ord a), puisque Ord « implique » Eq. Plus important encore, les méthodes pour les opérations des sous-classes peuvent supposer l'existence de méthodes définies pour les opérations des superclasses. Par exemple, la déclaration de Ord dans le Prélude Standard contient cette méthode par défaut pour < :

```
x < y                    =   x <= y && x /= y
```

Pour un exemple de l'utilisation de Ord, le typage principal de quicksort défini dans la section 2.4.1 est :

```
quicksort                :: (Ord a) => [a] -> [a]
```

En d'autres termes, quicksort n'opère que sur des listes de valeurs de type ordonné. Ce typage de quicksort est attribué en raison de l'utilisation des opérateurs de comparaison < et >= dans sa définition.

Haskell permet également l'« héritage multiple », puisque les classes peuvent avoir plus d'une superclasse. Par exemple, la déclaration :

```
class (Eq a, Show a) => C a where ...
```

créé une classe C qui hérite aussi bien des opérations de Eq que de celles de Show.

Les méthodes de classe sont traitées comme des déclarations de niveau supérieur dans Haskell. Elles partagent le même espace de nom que les variables ordinaires. Un même nom ne peut pas être utilisé pour dénoter à la fois une méthode de classe et une variable ou des méthodes dans différentes classes.

Les contextes sont également autorisés dans les déclarations de données (voir §4.2.1).

Les méthodes de classe peuvent contenir des contraintres supplémentaires sur n'importe quel type de variable à l'exception de ceux définissant la classe courante. Par exemple, dans cette classe :

```
class C a where
  m                      :: Show b => a -> b
```

la méthode m impose que le type b soit une instance de la classe Show. Cependant, la

méthode m ne peut contenir aucune autre contrainte de classe sur le type a. Celles-ci devraient plutôt faire partie du contexte dans la déclaration de classe.

À ce stade, nous avons utilisé les types de « premier-ordre ». Par exemple, le constructeur de type Arborescence a toujours été mis en parité avec un argument, comme dans Arborescence Integer (une arborescence contenant des valeurs entières Integer) ou Arborescence a (représentant la famille d'arborescences contenant des valeurs de type a). Mais Arborescence est elle même un constructeur de type, et en tant que tel, accepte un type en tant qu'argument et retourne un type en tant que résultat. Il n'existe pas de valeurs, dans Haskell, qui sont de ce type, mais de tels type d'« ordre-spérieur » peuvent être utilisés dans les déclarations de classe.

Pour commencer, considérons la classe Functor suivante (extraite du Prélude) :

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

La fonction fmap généralise la fonction map utilisée précédemment. Notez que la variable de type f est appliquée à d'autres types dans f a et f b. Nous pourrions donc nous attendre à ce que celle-ci puisse être liée à un type tel que Arborescence qui peut être appliqué à un argument. Une instance de Functor pour le type Arborescence serait :

```
instance Functor Arborescence where
  fmap f (Feuille x)       = Feuille   (f x)
  fmap f (Branche t1 t2) = Branche (fmap f t1) (fmap f t2)
```

Cette déclaratin d'instance déclare que Arborescence, plutôt que Arborescence a, est une instance de Functor. Cette capacité est très utile, et démontre ici la possibilité de décrire des types « conteneurs » génériques, permettant à des fonctions telles que fmap de fonctionner uniformément sur des arborescences arbitraires, des listes, et d'autres types de données.

[Les applications de types sont écrites de la même manière que les applications de fonctions. Le type T a b est traduit comme (T a) b. Les types tels que les tuples qui utilisent une syntaxe spéciale peuvent être écrits dans un style alternatif qui autorise la curyfication. Pour les fonctions, -> est un constructeur de type. Les types f -> g et (->) f g sont équivalents. De même, les types [a] et [] a sont équivalents. Pour les tuples, les constructeurs de type (ainsi que pour les constructeurs de données) sont (,), („), et ainsi de suite.]

Comme nous le savons, le système de typage détecte les erreurs de type dans les expressions. Mais qu'en est-il des erreurs dues à des expressions de type mal formées? L'expression (+) 1 2 3 provoque une erreur de type puisque (+) n'accepte que deux arguments. De même, le type Arborescence Int Int devrait produire une erreur puisque le type Arborescence n'accepte qu'un seul argument. Alors, comment Haskell détecte-t-il les expressions de type mal formées? La réponse est un deuxième système de typage qui

garantit que les types soient corrects! Chaque type a une « espèce » qui lui est associée ce qui garantit que les types sont utilisés correctement.

Les expressions de type sont classées selon différentes espèces qui prennent une des deux formes possibles :

* Le symbole ∗ représente l'espèce de type associée au objets de données concrètes. C'est à dire, si la valeur « v » est de type « t », l'espèce de « v » doit être ∗.
* Si κ1 et κ2 sont des espèces, alors κ1 → κ2 est l'espèce de types qui prennent un type d'espèce κ1 et retournent un type d'espèce κ2.

Le constructeur de type Arborescence est d'espèce ∗→∗. Le type d'Arborescence Int est d'espèce ∗. Les membres de la classe Functor doivent tous être de la même espèce ∗→∗. Une erreur d'espèce résulterait d'une déclaration telle que :

```
instance Functor Integer where ...
```

puisque l'espèce de Integer est ∗.

Les espèces n'apparaissent pas directement dans les programmes Haskell. Le compilateur infère les espèces avant de passer à la vérification des types sans avoir besoin d'une quelconque « déclaration d'espèce ». Les espèces restent dans les tréfonds des programmes Haskell sauf lorsqu'une signature de type éronnée conduit à une erreur d'espèce. Les espèces sont assez simples pour que les compilateurs soient capables de fournir des messages d'erreurs descriptifs quand des conflits d'espèce surviennent. Voir §4.1.1 et §4.6 pour plus d'informations sur les espèces.

Une perspective différente

Avant d'aller plus loin avec d'autres exemples de l'utilisation des classes de types, il peut être bon de se choisir deux autres points de vue sur les classes de types. Le premièr est par analogie avec la programmation orientée objet (POO). Dans l'énonciation suivante à propos de la POO, une simple substitution des classes de types par des classes, et des types par des objets, donne un bon résumé du méchanisme des classes de types de Haskell :

« Les classes renferment des jeux communs d'opérations. Un objet particulier peut être l'instance d'une classe, et aura une méthode correspondant à chaque opération. Les classes peuvent être arrangées hiérarchiquement, formant les notions de superclasses et sousclasses, et permettant l'héritage d'opérations/méthodes. Une méthode par défaut peut aussi être associée à une opération ».

Contrairement à la POO, il devrait être clair que les types ne sont pas des objets, et en praticulier il n'y a pas de notion d'état interne mutable des objets ou des types. Un avantage de Haskell par rapport à quelques langages POO est d'imposer que les méthodes

opèrent sur des types cohérents : Une tentative d'appliquer une méthode à une valeur dont le type n'appartient pas à la classe requise sera détectée à la compilation plutôt qu'à l'exécution. En d'autres termes, les méthodes ne sont pas « examinées » à l'exécution mais sont simplement passées en tant que fonctions d'ordre supérieur.

On aura encore une autre perspective en considérant la relation entre le polymorphisme paramétrique et ad hoc. Nous avons montré comment le polymorphisme paramétrique pouvait être utile dans la définition de familles de types en quantifiant universellement pour tous les types. Quelquefois, cependant, cette quantification universelle peut s'avérer trop large—nous souhaitons quantifier sur des ensembles plus restreints de types, tels que les types dont les éléments peuvent être comparés pour tester leur égalité. On peut se représenter les classes de types précisément comme une manière d'obtenir ceci. En fait, nous pouvons aussi nous représenter le polymorphisme paramétrique comme un genre de surcharge! C'est juste que la surcharge intervient implicitement sur tous les types plutôt que sur un ensemble restreint de types (c.-à.d. les classes de types).

Comparaisons avec d'autres langages

Les classes utilisées par Haskell sont similaires à celles utilisées dans d'autres langages orientés objet, tel que C++ et Java. Cependant, il y a d'importantes différences :

* Dans Haskell, la définition d'un type est séparée de la définition des méthodes associées à ce type. Normalement, une classe dans C++ ou Java, définit à la fois une structure de données (les variables membres) et les fonctions associées à cette structure (les méthodes). Dans Haskell, ces définitions sont séparées.
* Les méthodes de classe définies par une classe Haskell correspondent à une fonction virtuelle dans une classe C++. Chaque instance d'une classe fournit sa propre définition pour chaque méthode. Le comportement par défaut d'une classe correspond aux définitions par défaut pour une fonction virtuelle dans la classe de base.
* Les classes Haskell sont grossièrement similaires à des interfaces Java. De même qu'une déclaration d'interface, une déclaration de classe Haskell définit un protocole pour l'utilisation d'un objet plutôt que la définition de l'objet lui même.
* Haskell ne permet pas le style de surcharge C++ qui fonctionne sur différents types et partageant le même nom.
* Le type d'un objet Haskell ne peut pas être implicitement forcé. Il n'existe pas de classe de base univerelle telle que Object dont les valeurs peuvent être projetées dans ou « hors de ».
* C++ et Java attachent des informations d'identification (telle que VTable) à la représentation de l'objet à l'exécution. Dans Haskell, de telles informations sont attachées à des valeurs logiquement plutôt que physiquement, à travers le système de typage.
* Il n'y a pas de contrôle des accès (tels que les constituants de classe privée ou publique) intégrés au système de classes de Haskell. En lieu et place, le système de modules doit être utilisé pour cacher ou réveler les composants d'une classe.

VI. Les types, encore
VI-1. La déclaration newtype
VI-2. Les étiquettes de champs
VI-3. Les constructeurs stricts de données

VI. Les types, encore

Ici nous examinons quelques uns des aspects les plus avancés des déclarations de type.

VI-1. La déclaration newtype

Une pratique commune de programmation est de définir un type dont la représentation est identique à un type pré-existant mais qui a une identité différente dans le système de typage. Dans Haskell, la déclaration newtype crée un nouveau type à partir d'un type pré-existant. Par exemple, les nombres naturels peuvent être représentés par le type Integer en utilisant la déclaration suivante :

```
newtype Natural = MakeNatural Integer
```

Ceci créé un type entièrement nouveau, Natural, dont l'unique constructeur contient un simple Integer. Le constructeur MakeNatural opère une conversion entre un Natural et un Integer :

```
toNatural               :: Integer -> Natural
toNatural x | x < 0     = error "Erreur:ne peut créer de naturel négatif!"
            | otherwise = MakeNatural x

fromNatural                 :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

La déclaration d'instance suivante ajoute Natural à la classe Num :

```
instance Num Natural where
    fromInteger       = toNatural
    x + y             = toNatural (fromNatural x + fromNatural y)
    x - y             = let r = fromNatural x - fromNatural y
                        in
                          if r < 0 then error "Soustraction non naturelle"
                                   else toNatural r
    x * y             = toNatural (fromNatural x * fromNatural y)
```

sans cette déclaration, Natural ne serait pas inclus dans Num. Les instances déclarées pour l'ancien type ne sont pas reportées au nouveau type. En fait, le seul objectif de ce type est d'introduire une instance de Num différente ; ce qui ne serait pas possible si Natural était défini en tant que synonyme de Integer.

Cela fonctionnerait aussi en utilisant une déclaration data plutôt qu'une déclaration newtype. Cependant, la déclaration data est plus « coûteuse » pour la représentation de valeurs de type Natural. L'utilisation de newtype évite le niveau supplémentaire d'une

indirection (provoquée par la paresse) qu'une déclaration data nécessiterait. Voir la section 4.2.3 du Haskell Report pour plus d'informations sur la relations entre les déclaration newtype, data, et type. [À l'exception du mot-clef, la déclaration newtype utilise la même syntaxe qu'une déclaration data avec un unique constructeur contenant un unique champ , ce qui est logique puisque les types définis avec newtype sont presque identiques à ceux définis avec une déclaration ordinaire data.]

VI-2. Les étiquettes de champs

Les champs à l'intérieur d'un type de données Haskell sont accessibles soit par position soit par nom en utilisant les « étiquettes de champs ». Considérons un type de données pour un point dans le plan :

```
data Point = Pt Float Float
```

Les deux composants d'un Point sont les premier et deuxième arguments du constructeur Pt. Une fonction telle que

```
pointx                  :: Point -> Float
pointx (Pt x _)         =  x
```

peut être utilisée pour obtenir le premier composant d'un point d'une manière plus descriptive, mais, pour de larges structures, il devient laborieux de créer de telles fonctions à la main.

Les constructeurs dans une déclaration data peuvent être déclarés avec des « étiquettes de champs », entre crochets cursifs. Ces étiquettes de champs identifient les composants d'un constructeur par leur nom plutôt que par leur position. Voici une définition alternative de Point :

```
data Point = Pt {pointx, pointy :: Float}
```

Ce type de données est identique à la définition précédente de Point. Le constructeur Pt est le même dans les deux cas. Cependant, cette déclaration définit également deux noms de champs, pointx et pointy. Ces noms de champs peuvent être utilisés comme « fonctions de sélection » pour extraire un composant d'une structure. Dans cet exemple, les sélecteurs sont :

```
pointx                  ::   Point -> Float
pointy                  ::   Point -> Float
```

Voici une fonction qui utilise ces sélecteurs :

```
absPoint                :: Point -> Float
absPoint p              =  sqrt (pointx p * pointx p +
                                 pointy p * pointy p)
```

Les étiquettes de champs peuvent aussi être utilisées pour construire de nouvelles valeurs. L'expression Pt {pointx=1, pointy=2} est identique à Pt 1 2. L'utilisation de noms de champs dans la déclaration d'un constructeur de données n'empêche pas d'accéder aux champs par leur position ; aussi bien Pt {pointx=1, pointy=2} que Pt 1 2 sont autorisés. Lorsque l'on construit une valeur en utilisant les noms de champs, certains champs peuvent être omis ; les champs absents seront indéfinis.

La correspondance de motif utilisant les noms de champs utilise une syntaxe similaire pour le constructeur Pt :

```
absPoint (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
```

Une fonction de mise à jour utilise les valeurs de champs d'une structure existante pour compléter les composants d'une nouvelle structure. Si p est un Point, alors p {pointx=2} est un point avec le même pointy que p mais dont le pointx est remplacé par 2. Ce n'est pas une mise à jour destructive : la fonction de mise à jour crée simplement une nouvelle copie de l'objet, complétant les champs spécifiés avec de nouvelles valeurs.

[Les crochets cursifs utilisés en conjonction avec les étiquettes de champs sont un peu particuliers : la syntaxe Haskell permet, en principe, d'omettre les crochets en utilisant la règle de « mise en forme » (décrite dans la section 4.6). Cependant, les crochets cursifs associés aux noms de champs doivent être utilisés explicitement]

Les noms de champs ne sont pas restreints aux types comprenant un unique constructeur (appelés couramment types « enregistrement » [record en anglais]). Dans un type comprenant de multiples constructeurs, les opérations de sélection ou de mise à jour utilisant des noms de champs peuvent échouer à l'exécution. C'est le même comportement que la fonction head lorsqu'elle est appliquée à une liste vide.

Les étiquettes de champs partagent le même espace de nom de niveau supérieur avec les variables ordinaires et les méthodes de classe. Un nom de champs ne peut pas être utilisé dans plus d'un type de données dans son étendue. Cependant, à l'intérieur d'un type de données, le même nom de champ peut être utilisé dans plus d'un des constructeurs à condition qu'il ait le même typage dans tous les cas. Par exemple, dans ce type de données :

```
data T = C1 {f :: Int, g :: Float}
       | C2 {f :: Int, h :: Bool}
```

le nom de champ f s'applique à tous les constructeurs dans T. Donc, si x est de type T, alors x{f=5} fonctionnera pour les valeurs créées par l'un des deux constructeurs dans T.

Les noms de champs ne changent pas la nature élémentaire d'un type de données algébrique ; il s'agit simplement d'une syntaxe commode pour accéder aux composants

d’une structure de données par nom plutôt que par position. Ils rendent les constructeurs avec beaucoup de composants plus faciles à gérer étant donné que des champs peuvent être ajoutés ou retirés sans changer toutes les références à ces constructeurs. Pour tous les détails sur les étiquettes de champs et leur sémantique, voir section §4.2.1.

VI-3. Les constructeurs stricts de données

Les structures de données dans Haskell sont généralement « paresseuses » : les composants ne sont évalués que lorsque c’est nécessaire. Cela permet d’avoir des structures contenant des éléments qui, s'ils étaient évalués, provoqueraient une erreur ou ne se termineraient jamais. Les structures paresseuses de données améliorent l’expressivité de Haskell et représentent un aspect essentiel du style de programmation Haskell.

En interne, chaque champ d’un objet paresseux de données est emballé dans une structure communément désignée sous le nom de « pensée » (traduction très libre du mot anglais « thunk ») qui encapsule le calcul informatisé définissant la valeur du champ. Haskell n’entre dans cette pensée que lorsque la valeur est requise ; les pensées qui contiennent des erreurs (« ⊥ ») n’affectent pas les autres éléments d’une structure de données. Par exemple, le tuple ('a',_|_) est une valeur Haskell parfaitement légale. Le 'a' peut être utilisé sans déranger les autres composants du tuple. La plupart des langages de programmation sont « stricts » plutôt que paresseux : c’est à dire que tous les composants d’une structure de données sont réduits à des valeurs avant d’être placés dans la structure.

Il y a beaucoup de « surcoût » associés aux pensées : elles prennent du temps à construire et à évaluer, elles occupent de la place dans le tas, et elles causent une rétention d’autres structures nécessaires à l’évaluation de la pensée dans le ramasseur-de-miette. Pour éviter ces surcoûts, les fanions de rigueur (strictness flags en anglais) dans les déclarations data permettent à des champs spécifiques de constructeurs d’être évalués immédiatement, interdisant la paresse de manière sélective. Un champ marqué avec « ! » dans une déclaration de données est évaluée lorsque la structure est créée au lieu d’être mise en attente dans une pensée. Il y a beaucoup de situations dans lesquelles il est adéquat d’utiliser les fanions de rigueur :

* Les composants de structures qui devront dans tous les cas être évalués à un moment donné lors de l’exécution d’un programme.
* Les composants de structures qui sont simples à évaluer et qui ne provoquent jamais d’erreur.
* Les types dans lesquels des valeurs partiellement indéfinies n’ont pas de signification.

Par exemple, la bibliothèque des nombres complexes définit le type Complex comme ceci :

```
data RealFloat a => Complex a = !a :+ !a
```

[Notez le définition infixée du constructeur :+.] Cette définition marque les deux

composants, les parties réelle et imaginaire du nombre complexe, comme étant stricts. C'est une représentation plus compacte des nombres complexes, mais cela a un prix : un nombre complexe avec un composant indéfini, 1 :+ « ⊥ » par exemple, est totalement indéfini (« ⊥ »). Étant donné qu'il n'est pas utile de définir des nombres complexes partiellement définis, il est approprié d'utiliser les fanions de rigueur pour en obtenir une représentation plus efficace.

Les fanions de rigueur peuvent être utilisés pour traiter les fuites de mémoire : les structures retenues par le ramasseur-de-miette mais qui ne sont plus nécessaires aux calculs.

Le fanion de rigueur, !, ne peut apparaître que dans les déclarations data. Il ne peut pas être utilisé pour les autres signatures de types ni dans aucune autre définition de type. Il n'existe pas de fanion de rigueur équivalent pour marquer les arguments de fonctions, bien que le même effet puisse être obtenu en utilisant les fonctions seq ou !$. Voir §4.2.1 pour plus de détails.

Il est difficile de donner des consignes exactes pour l'utilisation des fanions de rigueur. Ils devraient être utilisés avec prudence : la paresse est une propriété fondamentale de Haskell et l'ajout de fanions de rigueur peut conduire à des boucles infinies difficiles à trouver ou avoir des conséquences inattendues.

VII. Entrées/Sorties
VII-1. Opérations d'E/S de base
VII-2. Programmer avec des actions
VII-3. Gestion des exceptions
VII-4. Fichiers, canaux et gestionnaires
VII-5. Haskell et la programmation impérative

VII. Entrées/Sorties

Le système d'entrées/sorties (input/output ou I/O en anglais) dans Haskell est purement fonctionnel, mais garde tout de la puissante expressivité que l'on trouve dans les langages de programmation conventionnels. Dans les langages impératifs, les programmes procèdent par « actions » qui examinent et modifient l'état courant. Des actions typiques incluent la lecture et l'écriture de variables globales, l'écriture de fichiers, la lecture des entrées, l'ouverture de fenêtres. De telles actions sont également intégrées dans Haskell mais sont proprement séparées du coeur purement fonctionnel du langage.

Le système d'entrées/sorties de Haskell est construit sur des fondations mathématiques quelque peu intimidantes : les « monades ». Cependant, il n'est pas nécessaire de comprendre la théorie des monades pour utiliser le système d'E/S qui repose sur cette théorie. Les monades sont une structure conceptuelle qui cadre bien avec le système d'E/S. Il n'est pas plus nécessaire de comprendre la théorie des monades pour réaliser des E/S Haskell que de comprendre la théorie des groupes pour réaliser de simples opérations arithmétiques. Une explication détaillée des monades se trouve en section 9.

Les opérateurs monadiques sur lesquels sont construits le système d'E/S ont également d'autres usages. Pour le moment, nous allons éviter le terme monade et nous concentrer sur l'utilisation du système d'E/S. Il est préférable de se représenter les monades d'E/S simplement comme un type de donnée abstrait.

Les actions sont définies, plutôt qu'invoquées, dans le langage d'expression de Haskell. L'évaluation de la définition d'une action ne déclenche pas l'action. En fait, l'invocation des actions se déroule hors de l'évaluation de l'expression que nous avons considérée jusqu'ici.

Les actions sont soit atomiques, telles que définies dans les primitives du système, soit une composition séquentielle d'autres actions. Les monades d'E/S contiennent des primitives qui construisent des actions composées, un procédé similaire à celui utilisé avec « ; » pour définir une séquence ordonnée d'instructions dans d'autres langages. Par conséquent, les monades servent de colle qui crée un lien entre les actions dans un programme.

VII-1. Opérations d'E/S de base

Toute action d'E/S retourne une valeur. Dans le système de typage, la valeur retournée est

« typée » E/S, ce qui distingue les actions des autres valeurs. Par exemple, le type de la fonction getChar est :

```
getChar                   ::   IO Char
```

Le IO Char indique que getChar, lorsqu'il est invoqué, exécute des actions qui retournent un caractère. Les actions qui ne retournent pas de valeurs dignes d'intérêt utilisent le type unitaire (). Par exemple, la fonction putChar :

```
putChar                   ::    Char -> IO ()
```

prend un caractère en tant qu'argument mais ne retourne rien d'utile. Le type unitaire est similaire à void dans d'autres langages.

Les actions sont mises en séquence en utilisant un opérateur dont le nom est quelque peu cryptique : »= (ou « lie »). Plutôt que d'utiliser cet opérateur directement, nous choisirons une friandise syntaxique, la notation do, pour cacher ces opérateurs de séquence sous une syntaxe qui ressemble plus aux langages conventionnels. La notation do peut facilement être étendue à »=, tel que décrit dans §3.14.

Le mot-clef do initie une séquence d'instructions qui sont exécutées dans l'ordre. Une instruction peut être une action, un motif lié au résultats d'une action en utilisant <-, ou un ensemble de définitions locales utilisant let ou where afin d'omettre les crochets cursifs et le point-virgule avec une indentation correcte. Voici un programme simple qui lit puis affiche un caractère :

```
main                      :: IO ()
main                      =  do c <- getChar
                                putChar c
```

L'utilisation du nom main est importante : main est défini pour être le point d'entrée d'un programme Haskell (similaire à la fonction main en C), et doit être de type IO, en général IO (). (Le nom main n'est spécial que dans le module Main; Nous en dirons plus sur les modules ultérieurement). Ce programme exécute deux actions à la suite : premièrement il lit un caractère qu'il lie à la variable c et ensuite il affiche ce caractère. Contrairement à une expression let ou les variables sont disponibles pour toutes les définitions cadrées par let, les variables définies par <- ne sont disponibles que dans les instructions suivantes.

Il y a encore une pièce manquante. Nous pouvons invoquer des actions et examiner leurs résultats en utilisant do, mais comment faire pour retourner une valeur à partir d'une séquence d'actions? Par exemple, considérons la fonction ready qui lit un caractère et retourne True si le caractère est un « y » :

```
ready                     :: IO Bool
ready                     =  do c <- getChar
                                c == 'y'       -- Mal!!!
```

Ceci ne fonctionne pas, parce que la deuxième instruction dans le cadre du do n'est qu'une valeur booléenne, et non pas une action. Nous devons prendre ce booléen et créer une action qui ne fait rien mais retourne le booléen en tant que résultat. C'est précisément ce que fait la fonction return :

```
return                  ::   a -> IO a
```

La fonction return complète la suite ordonnée de primitives. La dernière ligne de ready devrait donc être return (c == 'y'). Nous sommes maintenant prêts à aborder des fonctions d'E/S plus compliquées. Commençons par la fonction getLine :

```
getLine      :: IO String
getLine      =  do c <- getChar
                   if c == '\n'
                      then return ""
                      else do l <- getLine
                              return (c :l)
```

Notez le deuxième do dans la clause else. Chaque do initie une chaîne simple d'instructions. Toute construction intercalée, telle que le if, doit utiliser un nouveau do pour initialiser de nouvelles séquences d'actions.

La fonction return admet une valeur ordinaire telle qu'une booléenne dans le royaume des actions d'E/S. Et dans l'autre sens, qu'en est-il? Peut-on invoquer des actions d'E/S dans une expression ordinaire? Par exemple, comment pouvons nous exprimer x + print y dans une expression de façon à ce que y soit affiché quand l'expression est évaluée? La réponse est : c'est impossible! Il n'est pas possible de pénétrer discrètement dans le monde impératif alors que l'on est en plein milieu d'un code purement fonctionnel. Toute valeur « infectée » par l'univers impératif doit être signalée comme telle. Une fonction telle que

```
f    ::  Int -> Int -> Int
```

ne peut absolument pas réaliser d'E/S puisque IO n'apparaît pas dans le type de la valeur retournée. Cette contrainte est assez insupportable pour les programmeurs habitués à placer des instructions d'affichage librement dans leur code lors du débuggage. En fait, il existe quelques fonctions non-sûres qui permettent de contourner ce problème mais il est préférable de laisser cela aux programmeurs expérimentés. Les paquets de débuggage (comme Trace) font souvent une utilisation libérale de ces « fonctions interdites » d'une manière sécurisée.

VII-2. Programmer avec des actions

Les actions d'E/S sont des valeurs Haskell ordinaires : Elles peuvent être passées à des fonctions, placées dans des structures et utilisées comme n'importe quelle autre valeur Haskell. Considérons cette liste d'actions :

```
todoList :: [IO ()]

todoList = [putChar 'a',
            do putChar 'b'
               putChar 'c',
            do c <- getChar
               putChar c]
```

Cette liste n'invoque, en fait, aucune action — elle se contente simplement de les contenir. Pour réunir ces actions en une unique action, une fonction telle que sequence est requise : sequence_ est nécessaire :

```
sequence_        :: [IO ()] -> IO ()
sequence_ []     =  return ()
sequence_ (a:as) =  do a
                       sequence as
```

Ce que l'on peut simplifier si l'on retient que do x;y est interprétée comme x » y (voir section 9.1). La fonction foldr reproduit ce motif de récursion (voir le Prélude Standard pour une définition de foldr) ; une meilleure définition de sequence_ est :

```
sequence_        :: [IO ()] -> IO ()
sequence_        =  foldr (») (return ())
```

La notation do est un outil utile mais dans ce cas, l'opérateur monadique sous-tendu » est plus approprié. Une compréhension des opérateurs sur lesquels do est construit est assez utile au programmeur Haskell.

La fonction sequence_ peut être utilisée pour constuire putStr à partir de putChar :

```
putStr                :: String -> IO ()
putStr s              =  sequence_ (map putChar s)
```

Une des différences entre Haskell et la programmation impérative conventionnelle est visible dans putStr. Dans un langage impératif, mapper une version impérative de putChar sur la chaîne de caractères serait suffisant pour l'afficher. Dans Haskell, cependant, la fonction map n'exécute aucune action mais crée une liste d'actions, une par caractère dans la chaîne de caractères. L'opération foldr dans sequence_ utilise la fonction » pour combiner toutes les actions individuelles en une seule action. Le return () utilisé ici est tout à fait nécessaire – foldr a besoin d'une action nulle à la fin de la chaîne d'actions qu'elle créé (en particulier s'il n'y a pas de caractères dans la chaîne de caractères !).

Le Prélude Standard et les bibliothèques contiennent beaucoup de fonctions qui sont utiles pour mettre des action d'E/S en séquence. Celles-ci sont généralement généralisées en monades arbitraires ; toute fonction ayant un contexte incluant Monad m => fonctionne avec le type IO.

VII-3. Gestion des exceptions

À ce stade, nous avons évité le sujet des exceptions durant les opérations d'E/S. Qu'arriverait-il si getChar était confronté à une fin de fichier ? Nous utilisons le terme « erreur » pour « ⊥ » : une condition de laquelle on ne peut récupérer telle qu'une non-terminaison, ou une correspondance de motif négative. Les exceptions, au contraire, peuvent être appréhendées et traitées dans une monade d'E/S. Pour traiter les conditions exceptionnelles telle qu'un « fichier non trouvé » dans une monade d'E/S, un mécanisme de gestion est utilisé, dont les fonctionnalités sont similaires à celles du ML standard. Il n'y a pas de syntaxe ou de sémantique particulière ; la gestion des exceptions fait partie de la définition des opérations d'E/S mises en séquence.

Les erreurs sont encodées en utilisant un type de donnée spécial, IOError. Ce type représente toutes les exceptions possibles qui peuvent se produire dans une monade d'E/S. C'est un type abstrait : il n'y a pas de constructeur disponible pour l'utilisateur dans IOError. Les prédicats permettent de requérir les valeurs d'IOError. Par exemple, la fonction

```
          isEOFError       :: IOError -> Bool
```

détermine si une erreur a été provoquée par une fin de fichier. En faisant de IOError un type abstrait, de nouveaux types d'erreurs peuvent être ajoutés au système sans changements notables du type de donnée. La fonction isEOFError est définie dans une librairie séparée, IO, et doit être explicitement importée dans le programme.

Un gestionnaire d'exception est de type IOError -> IO a. La fonction catch associe un gestionnaire d'exception à une action ou à un ensemble d'actions :

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

Les arguments de catch sont une action et un gestionnaire. Si l'action se déroule avec succès, son résultat est retourné sans invoquer le gestionnaire. Si une erreur se produit, elle est passée au gestionnaire en tant que valeur de type IOError et l'action associée au gestionnaire est à son tour invoquée. Par exemple, cette version de getChar retourne une nouvelle ligne quand une erreur est rencontrée :

```
getChar'               :: IO Char
getChar'               =  getChar `catch` (\e -> return '\n')
```

C'est plutôt abrupte puisque toutes les erreurs sont traitées de la même manière. Si c'est uniquement une fin de fichier qui doit être reconnue, la valeur de l'erreur doit être requise:

```
getChar'::IO Char
getChar'= getChar `catch` eofHandler
          where   eofHandler e = if isEofError e then return  '\n'
                                  else ioError e
```

La fonction ioError utilisée ici lance une exception au prochain gestionnaire d'exception. Le type de ioError est

```
ioError                 :: IOError -> IO a
```

Ce qui est similaire à return sauf que cela transfert le contrôle au gestionnaire d'exception plutôt que de continuer avec l'action d'E/S suivante. Il est permis de faire des appels emboîtés à catch, ce qui produit des gestionnaires d'exception emboîtés. En utilisant getChar', nous pouvons redéfinir getLine pour démontrer l'utilisation de gestionnaires emboîtés :

```
getLine' :: IO String
getLine' = catch getLine'' (\err -> return ("Error : " ++ show err))
           where
                   getLine'' = do c <- getChar'
                                  if c == '\n' then return ""
                                               else do l <- getLine'
                                                       return (c :l)
```

Les gestionnaires d'erreur emboîtés permettent à getChar' d'appréhender une fin de fichier alors que toute autre erreur aboutit à une chaîne de caractères commençant par "Error : " en provenance de getLine'. Pour faciliter le travail, Haskell fournit un gestionnaire d'exception par défaut, au niveau le plus élevé d'un programme, qui affiche l'exception et termine le programme.

VII-4. Fichiers, canaux et gestionnaires

À coté des monades d'E/S et du mécanisme de gestion des exceptions qu'il fournit, les utilitaires du système d'E/S dans Haskell sont, pour la plupart, assez proches de ceux des autres langages. Beaucoup de ces fonctions se trouvent dans la librairie IO plutôt que dans le Prélude. Cette librairie doit donc être importée explicitement pour être dans l'étendue (les modules et l'importation sont abordés dans la section 11). De même, la plupart de ces fonctions se trouvent dans le Library Report plutôt que dans le Haskell Report principal.

L'ouverture d'un fichier crée un « gestionnaire » (de type Handle) à utiliser dans les transactions d'E/S. La fermeture du gestionnaire ferme le fichier associé :

```
type FilePath          =  String  -- path names in the file system
openFile               :: FilePath -> IOMode -> IO Handle
hClose                 :: Handle -> IO ()
data IOMode            =  ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Les gestionnaires peuvent aussi être associés à des « canaux » : des ports de communication qui ne sont pas directement attachés à des fichiers. Quelques gestionnaires de canaux sont prédéfinis, y compris stdin (l'entrée standard), stdout (la sortie standard), et stderr (l'erreur standard). Les opérations d'E/S au niveau du caractère comprennent hGetChar et hPutChar, qui prennent un gestionnaire en tant qu'argument. La fonction

getChar utilisée précédemment peut être redéfinie par :

```
getChar                = hGetChar stdin
```

Haskell permet également de retourner comme une chaîne de caractères unique le contenu intégral d'un fichier ou d'un canal :

```
getContents            :: Handle -> IO String
```

De manière pragmatique, il pourrait sembler que getContents doit lire immédiatement l'intégralité d'un fichier ou d'un canal, ce qui aurait un impact négatif sur la performance en terme de temps et de taille dans certaines circonstances. Cependant, ce n'est pas le cas. Le point clef est que getContents retourne une liste « paresseuse » (c.-à-d. non stricte) de caractères (rappelez-vous que les chaînes de caractères ne sont que des listes de caractères dans Haskell), dont les éléments sont lus « à la demande » comme pour toute autre liste. On peut attendre d'une implémentation de Haskell qu'elle implémente ce comportement « à la demande » en lisant le fichier un caractère à la fois lorsqu'ils sont requis dans le calcul informatisé.

Dans cet exemple, un programme Haskell copie un fichier sur un autre :

```
main = do fromHandle <- getAndOpenFile "Copy from : " ReadMode
          toHandle   <- getAndOpenFile "Copy to : " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."

getAndOpenFile          :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

En utilisant la fonction paresseuse getContents, le transfert à la mémoire de l'intégralité du contenu du fichier n'est pas requis en une seule fois. Si hPutStr choisit de mettre en tampon la sortie, en écrivant par blocs de taille fixe la chaîne de caractères, il n'y aura en mémoire qu'un seul bloc à la fois. Le fichier en entrée est fermé implicitement quand le dernier caractère a été lu.

VII-5. Haskell et la programmation impérative

Un dernier mot, la programmation d'E/S soulève une question importante : il y a suspicion de ressemblance avec la programmation impérative ordinaire. Par exemple, la fonction getLine :

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

exhibe une saisissante ressemblance avec le code impératif :

```
function getLine() {
  c := getChar();
  if c == '\n' then return ""
              else {l := getLine();
                    return c :l}}
```

Alors, finalement, est-ce que Haskell a simplement réinventé la roue impérative?

Oui, en quelque sorte. Les monades d'E/S constituent un petit sous-langage impératif à l'intérieur de Haskell, et par conséquent, les composants d'E/S d'un programme peuvent sembler similaires à du code impératif ordinaire. Mais il y a une différence importante : il n'y a pas de sémantique particulière avec laquelle l'utilisateur devrait jongler. En particulier, le raisonnement par équations dans Haskell n'est pas compromis. L'impression d'impératif que laisse le code monadique dans un programme n'amoindrit pas les aspects fonctionnels de Haskell. Un programmeur fonctionnel expérimenté devrait être capable de minimiser les composants impératifs d'un programme, en limitant les monades d'E/S à une quantité minime de mises en séquence à un haut niveau. Les monades séparent proprement les composants fonctionnels et impératifs d'un programme ; contrairement aux langages impératifs fournissant des sous-ensembles fonctionnels mais qui n'ont pas de barrière bien définie entre l'univers purement fonctionnel et impératif.

VIII. Standard Haskell Classes
VIII-1. Classes pour l'égalité et la relation d'ordre
VIII-2. La Classe Enumeration
VIII-3. Les Classes Read et Show
VIII-4. Les Instances dérivées

VIII. Standard Haskell Classes

Dans cette partie, nous allons présenter les classes de type standard pré-définies en Haskell. Nous avons toutefois choisi d'omettre les méthodes les moins intéressantes. Le rapport Haskell contient une description plus complète. Ainsi, quelques classes standard sont incluses dans les bibliothèques standards de Haskell. Elles sont décrites dans le rapport sur les librairies d'Haskell.

VIII-1. Classes pour l'égalité et la relation d'ordre

Les classes Eq et Ord ont déjà été rencontrées dans cet article. La définition de Ord dans le Prélude est plus complexe que la version simplifiée que nous avions présenté précédemment. En particulier, observez la méthode compare :

```
data Ordering           =  EQ | LT | GT
compare                 :: Ord a => a -> a -> Ordering
```

La méthode compare est suffisante pour définir toutes les autres méthodes (via les définitions par défaut) de cette classe, et dans le meilleur des cas de créer des instances de Ord.

VIII-2. La Classe Enumeration

La classe Enum possède un ensemble d'opérations qui gèrent le sucre syntaxique des suites arithmétiques. Par exemple, l'expression de suite arithmétique [1,3..] signifie enumFromThen 1 3 (cf 3.10) pour la traduction formelle). On peut maintenant voir que les expressions de suites arithmétiques peuvent être utilisées pour générer des listes de n'importe quel type qui puissent être une instance de Enum. Cela n'inclut pas seulement la majorité des types numériques, mais aussi Char. Par exemple, ['a'..'z'] définit la liste des lettres minuscules dans l'ordre alphabétique. Par ailleurs, les types énumérés définis par l'utilisateur, comme Color, peuvent facilement être généré par une déclaration d'instance de Enum. Ainsi :

```
[Red .. Violet] => [Red, Green, Blue, Indigo, Violet]
```

Remarquez qu'une telle suite est arithmétique dans le sens que l'incrément entre chaque valeur est constant, même si les valeurs ne sont pas des nombres. La majorité des types de Enum peuvent être associés à des entiers avec une précision fixée. Ainsi, on trouve les méthodes fromEnum et toEnum qui traduisent des données du type Int au type Enum, ou inversement.

VIII-3. Les Classes Read et Show

Les instances de la classe Show sont celles qui peuvent être converties en chaîne de caractères (typiquement pour des entrées/sorties). La classe Read fournit des opérations pour parser les chaînes de caractères pour obtenir les valeurs qu'elles représentent. La fonction la plus simple de la class Show est show :

```
show                        :: (Show a) => a -> String
```

Bien sûr, show prend n'importe quelle valeur d'un type adéquate et renvoie sa représentation sous forme de chaîne de caractères, comme dans show (2+2) qui renverra 4. Tant que cela fonctionne cela suffit, mais on aura besoin de produire des chaînes de caractères plus complexes qui peuvent représenter de nombreuses valeurs, telles :

"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++ "."

Au bout d'un moment, les concaténations deviendront quelque peu inefficaces. En particulier, on peut considérer une fonction qui représente des arbres binaires sous forme de chaînes de caractères (cf 2.2.1), avec des marques adéquates pour montrer les sous-arbres, et la séparation des branches droite et gauche (pourvu que le type de l'élément soit représentable sous forme de chaîne de caractères.

```
showTree              :: (Show a) => Tree a -> String
showTree (Leaf x)     =  show x
showTree (Branch l r) =  "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Etant donné que (++) a une complexité temporelle linéaire en la longueur de son argument de gauche, showTree est potentiellement quadratique en la taille de l'arbre.

Pour restaurer la complexité linéaire, la fonction show est fournie :

```
shows                       :: (Show a) => a -> String -> String
```

shows prend en arguments une valeur affichable et une chaîne de caractères, et renvoie cette chaîne de caractères avec la représentation de la valeur concaténée au début. Le second argument sert comme une sorte d'accumulateur, et show peut maintenant être définie comme shows avec un accumulateur vide. C'est d'ailleurs la définition par défaut de show dans la classe Show :

```
show x                      =  shows x ""
```

On peut utiliser shows pour définir une version plus efficace de showTree, qui utilise aussi un argument comme accumulateur :

```
showsTree               :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s    =  shows x s
showsTree (Branch l r) s=  '<' : showsTree l ('|' : showsTree r ('>' : s))
```

Cela résout notre problème d'efficacité (showTree est désormais linéaire), mais la présentation de cette fonction (et d'autres du même style) peut être améliorée. Tout d'abord, créons un type synonyme :

```
type ShowS              =  String -> String
```

Il s'agit du type de la fonction qui retourne une chaîne de caractères représentant quelque chose, suivi d'une chaîne de caractères d'accumulation. Ensuite, on peut éviter de traîner ces accumulateurs, et ainsi éviter d'empiler les parenthèses à la droite des constructions un peu longues, en utilisant la composition fonctionnelle :

```
showsTree            :: (Show a) => Tree a -> ShowS
showsTree (Leaf x)    = shows x
showsTree (Branch l r)= ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

Lors de cette transformation est apparue quelque chose de plus important qu'une simple vérification du code : on est passé d'une représentation d'un niveau objet (ici des chaînes de caractères), et un niveau fonctionnel. On peut penser au typage comme si showTree mettait en relation un arbre et une fonction représentative. Des fonctions comme ('<' :) ou ("a string" ++) sont des fonctions de représentation primitives, et on peut à partir d'elle construire des fonctions plus complexes par composition.

Maintenant que l'on a transformé les arbres en chaînes de caractères, considérons le problème inverse. L'idée de base est le parsage pour un type a, ce qui demande une fonction prenant en argument une chaîne de caractères, et renvoyant une liste de paires (a, String). Le Prélude fournit un type synonyme pour de telles fonctions :

```
type ReadS a            =  String -> [(a,String)]
```

Normalement, le parseur renvoie une liste à un élément, qui contient la valeur de type a qui a été lue depuis la chaîne de caractères en entrée, et la chaîne de caractères restante une fois que cet élément est parsé. Cependant, si aucune lecture n'est possible, la liste retournée sera vide, ou s'il y a ambiguïté (plusieurs interprétations possibles), la liste renvoyée contiendra plus d'une paire. La fonction standard reads est un parseur pour n'importe quelle instance de la classe Read :

```
reads                   :: (Read a) => ReadS a
```

On peut utiliser cette fonction pour définir une fonction de parsage d'une chaîne de caractères représentant des arbres binaires, produite par showTree. Les listes en compréhension nous donnent une syntaxe pratique pour construire de tels parseurs :
(Une approche plus élégante consisterait à parser au moyen de monades et de combinateurs de parseurs. C'est d'ailleurs la méthode utilisée dans les bibliothèques standards de parsage fournies dans la majorité des systèmes Haskell)

```
readsTree               :: (Read a) => ReadS (Tree a)
```

```
readsTree ('<':s)        =  [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                                (r, '>':u) <- readsTree t ]
readsTree s              =  [(Leaf x, t)     | (x,t)      <- reads s]
```

Prenons un moment pour examiner cette définition de fonction en détail. Il y a deux cas principaux à considérer. Si le premier caractère de la chaîne de caractères à parser est '<', on devrait avoir une représentation d'une branche ; sinon on devrait avoir la représentation d'une feuille.
Dans le premier cas, il faut faire un appel récursif sur la chaîne de caractères restante après le <, le seul parsage possible devrait alors être un arbre Branch l r avec une chaîne de caractères résiduelle u, et respectant les conditions suivantes :

* L'arbre l peut être parsé à partir du début de la chaîne de caractères s ;
* La chaîne de caractères restante (qui suit la représentation de l) commence par '|'. Nous appellerons la queue de cette chaîne de caractères t ;
* L'arbre r peut être parsé depuis le début de t ;
* La chaîne de caractères restante après le parsage commence par '>', et u est sa queue.

Remarquez le pouvoir d'expression que l'on obtient en combinant de la reconnaissance de motifs et des listes par compréhension. La forme du résultat du parsage est donnée par l'expression principale de la liste par compréhension, les deux premières conditions sont exprimées par le premier générateur ("(l, '|':t) qui renvoie la liste des parsages de s, et les conditions restantes sont exprimées par le second générateur.

La seconde équation définie dit juste que pour parser la représentation d'une feuille, on parse la représentation du type de l'élément de l'arbre, et qu'on applique le constructeur Leaf à la valeur obtenue.

Pour l'instant, on considérera comme acquis le fait qu'il existe une instance de Read (et de Show) pour les types Integer (parmi les types couramment utilisés), qui fournit reads et ce comporte comme on l'attend, c'est-à-dire :

(reads "5 golden rings") :: [(Integer,String)] => [(5, " golden rings")]

Avec cela, le parseur devrait vérifier les évaluations suivantes :

readsTree "<1|<2|3>>" => [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree "<1|2" => []

Il y a deux problèmes dans notre définition de readsTree. D'un part, le parseur est un peu trop rigide. Il n'autorise pas les espaces avant ou entre les éléments de la représentation de l'arbre. D'autre part, la façon dont on parse les symboles de ponctuation est différente de celle dont on parse les valeurs des feuilles et les sous-arbres. Ce manque d'homogénéité ne facilite pas la lecture de la définition de la fonction. On peut résoudre ces deux problèmes en utilisant l'analyseur lexical fourni par le Prélude :

```
lex                          :: ReadS String
```

lex renvoie normalement une liste à un élément contenant une paire de chaînes de caractères : le premier lexème de la chaîne de caractères passée en argument, et le reste. Les règles lexicales sont celles des programmes Haskell, incluant les commentaires, que lex va ignorer, et autorisant les espaces. Si la chaîne de caractères en entrée est vide ou ne contient que des espaces et des commentaires, lex renverra [("","")]. Si elle n'est pas vide dans ce sens, mais qu'elle ne commence pas par un lexème valide après un nombre quelconque d'espaces et de commentaires, lex renverra [].

En utilisant l'analyseur lexical, notre parseur d'arbre ressemble désormais à cela :

```
readsTree        :: (Read a) => ReadS (Tree a)
readsTree s    =  [(Branch l r, x) | ("<", t) <- lex s,
                                     (l,   u) <- readsTree t,
                                     ("|", v) <- lex u,
                                     (r,   w) <- readsTree v,
                                     (">", x) <- lex w         ]
                  ++
                  [(Leaf x, t)     | (x,   t) <- reads s       ]
```

On peut désormais souhaiter utiliser readsTree et showsTree pour déclarer (Read a) => Tree a une instance de Read et (Show a) => Tree a une instance de Show. Cela nous permettrait d'utiliser des surcharges des fonctions génériques du Prélude pour parser et afficher les arbres. Par ailleurs, on pourrait automatiquement parser et afficher de nombreux autres types contenant des arbres, comme par exemple [Tree Integer]. A cet instant, readsTree et showsTree semblent être des candidats sérieux pour être les méthodes de Read et Show. Les méthodes showsPrec et readsPrec sont des versions paramétriques de shows et reads. Un paramètre supplémentaire est le niveau de priorité, qui peut servir pour mettre correctement les expressions contenant des constructeurs infixes entre parenthèses. Pour des types comme Tree, la notion de priorité peut être ignorée. Ainsi, les instances de Show et Read sont :

```
instance Show a => Show (Tree a) where
    showsPrec _ x = showsTree x

instance Read a => Read (Tree a) where
    readsPrec _ s = readsTree s
```

D'un autre point de vue, on peut aussi définir cette instance de Show à partir de showTree:

```
instance Show a => Show (Tree a) where
   show t = showTree t
```

Cependant, cela serait moins performant que la version ShowS. Remarquez que la classe Show définit les méthodes par défaut pour showsPrec et show, ce qui permet à l'utilisateur de redéfinir l'une d'elles dans la déclaration d'instance. Comme celles par défaut sont mutuellement récursives, une déclaration d'instance qui ne définit aucune

d'entre elles bouclera infiniment lors d'un appel. D'autres classes comme Num ont également ce problème d'interblocage par défaut.

On renverra les lecteurs intéressés au §D pour plus de détails sur Read et Show.

On peut tester les instances de Read et Show en appliquant (read . show) (qui représente la fonction identité) à quelques arbres, où read est une méthode spécialisée dérivée de reads :

```
read                        :: (Read a) => String -> a
```

Cette fonction échoue s'il n'existe pas d'unique parseur ou si l'entrée contient n'importe quoi en plus qu'une représentation d'une valeur du type a (et d'éventuels espaces et commentaires).

VIII-4. Les Instances dérivées

Rappelez-vous de l'instance de Eq pour les arbres que l'on a présenté dans la section 5 comme une déclaration simple (et ennuyante) à produire. On avait besoin que le type d'élément pour les feuilles soit un type "égalable". Ainsi, deux feuilles sont égales si elles contiennent des éléments égaux, et deux branches sont égales si leurs arbres gauche et droit sont respectivement égaux. Tous les couples d'arbres étant différents.

```
instance  (Eq a) => Eq (Tree a)  where
     (Leaf x)     == (Leaf y)         =  x == y
     (Branch l r) == (Branch l' r')   =  l == l' && r == r'
     _            == _                =  False
```

Heureusement, on n'a pas besoin d'aller le détail à chaque fois que l'on veut des opérateurs d'égalité pour un nouveau type. L'instance de Eq peut les dériver automatiquement à partir de la déclaration de données que l'on a spécifiée :

```
data  Tree a =  Leaf a | Branch (Tree a) (Tree a)  deriving Eq
```

La clause de dérivation produit implicitement une déclaration d'instance de Eq comme dans la section 5. Les instances de Ord, Enum, Read et Show peuvent aussi être générées par la clause de dérivation.
Plus d'un nom de classe peut être spécifié. Dans ce cas, la liste des noms doit être parenthèser et les noms séparés par des virgules.

Une instance dérivée de Ord pour Tree est légèrement plus compliquée que l'instance de Eq :

```
instance  (Ord a) => Ord (Tree a)  where
     (Leaf _)     <= (Branch _)     =  True
     (Leaf x)     <= (Leaf y)       =  x <= y
     (Branch _)   <= (Leaf _)       =  False
     (Branch l r) <= (Branch l' r')=  l == l' && r <= r' || l <= l'
```

Cela spécifie un ordre lexicographique. Les constructeurs sont ordonnés dans l'ordre de leur apparition dans la déclaration de données, et les arguments d'un constructeur sont comparés de la gauche vers la droite. Rappelez-vous que le type liste "natif" est sémantiquement équivalent à un type à deux constructeurs. En fait, voici sa déclaration complète :

```
data [a]        = [] | a : [a] deriving (Eq, Ord)      -- pseudo-code
```

(Les listes ont ainsi des instances de Show et Read, qui ne sont pas dérivées.) Les instances dérivées de Eq et Ord pour les listes sont celles par défaut. En particulier, les chaînes de caractères, considérées comme des listes de caractères, sont ordonnées comme il est prévu par le type sous-jacent Char, avec une sous-chaîne initiale comparant moins qu'une chaîne de caractères plus longue, comme par exemple "cat" < "catalog".

Dans la pratique, les instances de Eq et Ord sont toujours dérivées, et non redéfinies par l'utilisateur. En fait, on devrait fournir nos propres définitions des relations d'égalité et d'ordre de prédicats seulement avec quelques changements, en étant attentif au fait de maintenir les propriétés algébriques attendues de relation d'équivalence, et d'ordre total. Par exemple, un prédicat non-transitif (==) peut être désastreux, et troubler les personnes qui relisent le programme et qui confondraient les transformations manuelles et automatiques se basant sur le fait que le prédicat (==) est une approximation de l'égalité. Néanmoins, il est parfois nécessaire de fournir des instances de Eq et Ord différentes de celles qui auraient été dérivées. L'exemple le plus important est probablement le cas où un type de données abstrait pouvant se concrétiser en différents types ; ainsi plusieurs valeurs réelles distinctes peuvent avoir le même représentant abstrait.

Un type énuméré peut avoir une instance dérivée de Enum, et ici encore, un ordre défini par les constructeurs dans la déclaration de données. Par exemple :

```
data Day = Sunday | Monday | Tuesday | Wednesday
          | Thursday | Friday | Saturday         deriving (Enum)
```

Ici quelques exemples simples utilisent les instances dérivées pour ce type :

```
[Wednesday .. Friday ]    => [Wednesday, Thursday, Friday]
[Monday, Wednesday ..]    => [Monday, Wednesday, Friday]
```

Les instances dérivées de Read (ou Show) sont possibles pour tous les types dont les types composant ont des instances de Read (ou Show). (Les instances de Read et Show pour les types standards sont pour la plupart d'entre eux définis dans le Prélude. Quelques types, tels le type fonction (->), ont une instance de Show mais pas d'instance de Read correspondante.) La représentation textuelle définie par une instance dérivée de Show est cohérente avec la représentation des expressions Haskell constantes du type en question. Par exemple, si on ajoute Show et Read à la clause de dérivation pour le type Day, on obtiendrait:`show [Monday .. Wednesday] => "[Monday,Tuesday,Wednesday]"`

IX. A propos des monades
IX-1. Les classes monadiques
IX-2. Les monades intégrées
IX-3. Utilisation des monades

IX. A propos des monades

De nombreux débutants en Haskell sont déroutés par le concept de monades. On est pourtant fréquemment amené à les croiser, le système d'entrée/sortie ayant été construit autour d'une monade. D'ailleurs, une syntaxe spéciale a été conçue pour utiliser les monades (do expression), et la bibliothèque standard contient un module entièrement voué aux monades. Dans cette partie, nous rentrerons plus en détails dans la programmation avec monades, ce qui fait certainement d'elle la moins "gentille" de ce guide. En effet, nous ne chercherons pas seulement à regarder les caractéristiques du langage utilisant des monades, mais nous essaierons aussi de comprendre pourquoi les monades constituent un outil si puissant, et comment on peut les utiliser.
Il existe différentes manières d'aborder les monades, vous pourrez trouver plus d'explications sur haskell.org. Une autre bonne introduction pour l'utilisation pratique des monades est le guide de Wadler, Monads for Functional Programming.

IX-1. Les classes monadiques

Le préambule contient un certain nombre de classes définnisant les monades, qui sont utilisées en Haskell. Ces classes sont basées sur la construction des monades à partir de la théorie des catégories. Bien que la terminologie de la théorie des catégories fournit des noms pour les classes monadiques et pour les opérations, il n'est pas nécessaire de trop rentrer dans une telle abstraction mathématique pour avoir une compréhension intuitive de l'utilisation des monades.

Une monade est construite par-dessus un type polymorphe, comme IO. La monade en elle-même est définie comme une suite de déclarations associant un type avec une ou plusieurs classes monadiques, Functor, Monad et MonadPlus. Aucune de ses classes monadiques n'est dérivable.
En plus de IO, deux autres types présents dans le préambule sont des monades : les listes ([]) et Maybe.

Mathématiquement, les monades sont régies par un ensemble de règles qui devraient faire office d'opérations. Cette idée de règles n'est pas une spécificité des monades, Haskell contient d'autres opérations qui sont en fait régies par des règles. Par exemple, x /= y et not (x == y) devraient être les mêmes quel que soit le type des variables comparées. Cependant, on n'en a aucune garantie : les opérateurs == et /= sont des méthodes distinctes de la classe Eq, et on ne peut garantir un lien entre elles de cette manière.
Dans la même logique, les règles régissant les monades présentées ici ne sont pas imposées par Haskell, mais sont pourtant respectées par toutes les instances des classes

monadiques. Les règles monadiques donnent donc un indice sur la structure sous-jacente des monades ; en les examinant, on peut donc acquérir une intuition suffisante pour utiliser les monades.

La classe Functor, dont on a déjà parlé dans la partie V, définit une seule opération : fmap. Cette fonction applique une opération aux éléments d'un conteneur, et renvoie un conteneur avec les valeurs calculées. On peut d'ailleurs voir les types polymorphes comme des conteneurs de variables d'un autre type.
Ces lois régissent la méthode fmap de la classe Functor.

fmap id = id
fmap (f . g) = fmap f . fmap g

Ces règles assurent que la structure du conteneur reste inchangée après le passage de fmap, et que son contenu n'est pas réordonné.
La classe Monad définit deux opérateurs de base: >>= (bind) et return.

```
infixl 1  >>, >>=
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

    m >> k           =  m >>= \_ -> k
```

Les opérations bind, >>> et >>=, combinent deux valeurs monadiques, tandis que l'opération return injecte une variable dans la monade (le conteneur).
La signature de >>= nous permet de mieux la comprendre : ma >>= \v -> mb combine la valeur monadique ma contenant des variables de type a, et une fonction qui prend en argument une variable v de type a pour renvoyer la valeur monadique mb. Le résultat doit alors combiner ma et mb dans une monade contenant des variables de type b.
L'opérateur >> est utilisé lorsqu'on n'a pas besoin de la variable produite par le premier opérateur.

Une définition plus précise du binding dépend, bien entendu, des monades. Par exemple, dans la monade IO, x >>= y effectue deux actions séquentiellement, en passant le résultat de la première à la seconde. Pour les autres monades implémentées, les listes et le type Maybe, ces opérations monadiques peuvent être vues comme le passage de zéro variable, ou plus, d'une exécution à la suivante. On va bientôt montrer quelques exemples.
La syntaxe do permet une prise en main facile des chaînes d'opérations monadiques. Sa signification peut se résumer avec les deux règles suivantes :

```
do e1 ; e2      =        e1 >> e2
do p <- e1; e2  =        e1 >>= \p -> e2
```

Quand le motif présent dans la seconde règle est "réfutable", on peut appeler l'opération fail. Cela permet d'indiquer une erreur (comme dans la monade IO), ou retourner la valeur zéro (comme dans la monade liste). Par conséquent, une traduction plus complexe de cette opération pourrait être :

```
do p <- e1; e2 = e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

où "s" est une chaîne de caractères signalant l'emplacement de la déclaration do pour pouvoir l'utiliser dans un message d'erreur. Par exemple, dans la monade IO, une action telle que 'a' <- getChar appelera fail si le caractère saisi n'est pas 'a'. Cela va, à son tour, terminer brutalement le programme car le fail d'IO appelle la fonction error.
Les règles qui régissent >>>= et return sont

```
return a >>= k            =     k a
m >>= return              =     m
xs >>= return . f         =     fmap f xs
m >>= (\x -> k x >>= h)   =     (m >>= k) >>= h
```

La classe MonadPlus est utilisée pour les monades qui possèdent un élément zero et un opérateur plus.

```
class  (Monad m) => MonadPlus m  where
    mzero             :: m a
    mplus             :: m a -> m a -> m a
```

L'élément zero obéit aux règles suivantes :

```
m >>= \x -> mzero   =   mzero
mzero >>= m         =   mzero
```

En ce qui concerne les listes, la valeur nulle est [], la liste vide. La monade IO ne contient pas d'élément nul, et n'est pas pas un membre de cette classe...
Les règles régissant l'opérateur mplus sont :

```
m `mplus` mzero   =   m
mzero `mplus` m   =   m
```

L'opérateur mplus de la monade liste est la concaténation standard des listes.

IX-2. Les monades intégrées

Avec les opérations monadiques et les règles qui les régissent, que pouvons-nous faire ? Nous avons déjà étudié la monade IO en détail, nous allons donc nous intéresser aux deux autres monades intégrées à Haskell.
Pour les listes, le binding associe un ensemble de calculs pour chaque valeur dans la liste. Quand on l'utilise avec les listes, la signature de >>= devient :

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

Etant données une liste d'éléments de type a, et une fonction associant à une variable de type a une liste d'éléments de type b, binding applique cette fonction à chaque élément de type a contenu dans la variable d'entrée, et renvoie la liste contenant les valeurs de type b générées. La fonction return créera une liste à un seul élément. Ces opérations devraient nous sembler familières, car les opérations sur les listes peuvent facilement être exprimées sous forme d'opérations monadiques définies sur les listes. Par exemple, voici trois expressions distinctes représentant la même chose :

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

Cette définition dépend de la définition du fail dans cette monade, comme étant la liste vide. Pour résumer, chaque <- génère un ensemble de valeurs qui est passé en argument à l'exécution de la monade. Par conséquent, x <- [1,2,3] appelle trois fois l'exécution de la monade, une fois par élément de la liste. L'expression renvoyée, (x,y), sera évaluée pour toutes les combinaisons possibles de binding. Dans cette optique, la liste monade peut être vue comme des fonctions à nombre d'arguments variable. Par exemple, la fonction suivante

```
mvLift2              :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y        =  do x' <- x
                           y' <- y
                           return (f x' y')
```

transforme une fonction standard à deux arguments en une fonction à nombres d'arguments variable (grâce à une liste d'arguments), renvoyant la valeur pour toutes les combinaisons possibles de deux arguments d'entrée. Par exemple,

```
mvLift2 (+) [1,3] [10,20,30]       =>   [11,21,31,13,23,33]
mvLift2 (\a b->[a,b]) "ab" "cd"    =>   ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []             =>   []
```

Cette fonction est une version spécialisée de la fonction LiftM2 de la classe Monad. On peut la voir comme une fonction injectant une fonction quelconque dans la monade des listes, donc avec un nombre d'arguments en entrée variable.
La monade Maybe est similaire la monade listes : la valeur Nothing équivaut à [] et Just x équivaut à [x].

IX-3. Utilisation des monades

Attention il ne suffit pas de décrire les opérateurs monadiques et les règles qui les régissent, pour décrire les domaines où il est utile d'utiliser les monades. On peut voir qu'elles apportent une certaine modularité. En définissant une opération monadique, on peut construire tout ce qui est nécessaire pour intégrer de nouvelles fonctionnalités aux monades, et ce de manière transparente. L'article de Wadler est d'ailleurs un excellent exemple de l'utilisation des monades pour obtenir un programme modulaire.

On va commencer par utiliser une monade décrite de cet article, la state monad, pour construire une monade plus complexe autour d'un type state S, ressemblant à cela :

```
data SM a = SM (S -> (a,S))  -- The monadic type

instance Monad SM where
  -- defines state propagation
  SM c1 >>= fc2         =  SM (\s0 -> let (r,s1) = c1 s0
                                             SM c2 = fc2 r
                                         in c2 s1)
  return k              =  SM (\s -> (k,s))

-- extracts the state from the monad
readSM                  :: SM S
readSM                  =  SM (\s -> (s,s))

 -- updates the state of the monad
updateSM                :: (S -> S) -> SM ()  -- alters the state
updateSM f              =  SM (\s -> ((), f s))

-- run a computation in the SM monad
runSM                   :: S -> SM a -> (a,S)
runSM s0 (SM c)         =  c s0
```

Cet exemple définit un nouveau type, SM, pour être une structure qui contient implicitement un type S. Une structure du type SM t définit donc une variable de type t, qui pourra accéder en lecture et en écriture à l'état du type S. La définition de SM consiste simplement en celle de fonctions qui prennent en argument un état, et qui produisent deux résultats : la valeur renvoyée (de n'importe quel type), et un état mis à jour. Ici, on ne peut pas utiliser un type "synonyme", car on a besoin du nom du type comme SM, qui pourra être utilisé dans les déclarations instance. La déclaration newtype est souvent utilisée à la place de data.

La déclaration instance définit la "tuyauterie" de la monade : comment ordonnancer deux exécutions et la définition d'une exécution vide. L'ordonnancement (avec l'opérateur >>=) définit une exécution (notée par le constructeur SM) passant un état initial, s0, dans c1, et donnant le résultat de ce calcul, r, à c2, et renvoyant finalement le résultat de c2.

La définition de return est plus facile : il ne change pas d'état du tout... il sert juste à insérer

une variable dans une monade.

Tandis que >>= et return sont de simples opérations monadiques de séquentialisation, on a aussi besoin de primitives monadiques. Une primitive monadique est juste une opération qui est utilisée dans l'abstraction d'une monade et qui sert à "passer les vitesses" durant le travail de la monade. Par exemple, dans la monade IO, des opérateurs comme putChar sont des primitives puisque elles concernent le fonctionnement interne de la monade IO. De la même manière, la monade que nous sommes en train de présenter, utilise deux primitives : readSM et updateSM. Remarquez qu'elles dépendent de la structure interne de la monade (un changement de la définition de SM obligerait à modifier ces primitives).

Les définitions de readSM et updateSM sont simples : readSM exporte l'état de la monade pour pouvoir l'observer, alors que updateSM autorise l'utilisateur à modifier l'état de la monade. (On aurait aussi pu définir writeSM comme une primitive, mais updateSM semble la manière la plus naturelle d'effectuer une telle opération).

Finalement, on a besoin d'une fonction qui lance l'exécution de la monade, runSM. Elle prend en argument l'état initial et une séquence d'instructions, et donne à la fois la valeur renvoyée et l'état final.

D'un point de vue plus général, ce qu'on essaie de faire est de définir un exécution globale d'une série d'étapes (fonctions du type SM a), ordonnancées à l'aide de >>= et return. Ces étapes peuvent interagir avec l'état (via readSM ou updateSM), ou l'ignorer. Cependant, l'utilisation (ou la non-utilisation) de cet état est cachée : la forme des appels de fonctions ne diffère pas.

Plutôt que de présenter plusieurs petits exemples utilisant juste cette state monade, on va s'intéresser à un exemple plus complexe qui utilise, entre autres, la state monade. On définit un petit langage pour effectuer des calculs sur des ressources.
Pour cela, on construit un langage dans le but d'implémenter un sous-ensemble des types et des fonctions de Haskell. De tels langages utilisent les outils de base d'Haskell, types et fonctions, afin de construire une bibliothèque d'opérations et de types spécialement adaptée à nos besoins.

Dans cet exemple, on considère une exécution qui nécessite des "ressources". Si cette ressource est indisponible, l'exécution est suspendue. On utilise le type R pour signaler une commande utilisant les ressources contrôlées par notre monade. La définition de R est:

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

Chaque exécution est une fonction qui prend en arguments les ressources disponibles, qui renvoie les ressources restantes, couplée à un éventuel résultat, du type a, ou à une exécution suspendue, du type R a, qui contient tout le travail fait jusqu'à l'épuisement des ressources nécessaires.

```
instance Monad R where
  R c1 >>= fc2  = R (\r -> case c1 r of
                            (r', Left v)-> let R c2 = fc2 v
                                           in  c2 r'
                            (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
  return v      = R (\r -> (r, (Left v)))
```

Le type Resource est utilisé de la même manière que l'état à l'intérieur de la state monade. Cette définition se voit comme : pour effectuer deux exécutions nécessitant des ressources, c1 et fc2 (une fonction produisant c2), on donne les ressources initiales à c1. Le résultat sera :

* une valeur, v, et les ressources restantes, qui seront utilisées pour le lancement de la commande suivante (l'appel fc2 v)
* une suspension d'exécution, pc1, et les ressources restantes au moment de la suspension

La suspension doit prendre en compte la seconde commande à exécuter : pc1 suspend la première exécution, c1, on doit donc propager cette information à c2 afin de produire la suspension de l'exécution dans son ensemble. La définition de return laisse les ressources inchangées, tout en insérant v dans la monade.

La déclaration d'instance définit la structure de base de la monade, mais ne dit pas comment les ressources seront utilisées. La monade peut être utilisée pour contrôler de nombreux types de ressources, ou pour implémenter de nombreuses politiques d'utilisation des ressources. On va donner un exemple très simple de définition de ressources, en choisissant que Resource soit un Integer représentant les étapes disponibles de l'exécution.

```
type Resource           =  Integer
```

Cette fonction prend une étape à moins qu'aucune ne soit disponible.

```
step     :: a -> R a
step v =  c where
              c = R (\r -> if r /= 0 then (r-1, Left v)
                                     else (r, Right c))
```

Les constructeurs Left et Right font partie du type Either. Cette fonction poursuit l'exécution dans R en renvoyant v tant qu'il reste au moins une étape de l'exécution dont les ressources nécessaires sont encore disponibles. Si plus aucune étape n'est disponible, la fonction step suspend l'exécution courante (la suspension est capturée par c), et range cette exécution suspendue dans la monade.

A partir de là, on dispose des outils définissant une suite d'exécutions utilisant des ressources (les monades), et on peut exprimer une sorte d'utilisation des ressources avec

step. Enfin, on a besoin d'indiquer comment les exécutions dans cette monade seront exprimées.
On considérera une fonction d'incrémentation dans notre monade.

```
inc    :: R Integer -> R Integer
inc i  =  do iValue <- i
             step (iValue+1)
```

Cela définit l'incrémentation comme une simple étape d'exécution. Le <- est nécessaire pour sortir de la monade la valeur en argument. Le type de iValue est Integer au lieu de R Integer.

Toutefois, cette définition n'est pas particulièrement satisfaisante, en comparaison aux définitions standards des fonctions d'incrémentation. Ne pourrait-on pas, à la place, surcharger des opérateurs existants, comme le +, afin qu'ils puissent travailler dans le monde des monades ? On va commencer avec un ensemble de fonctions de lifting. Celles-ci importent des fonctionnalités existantes dans les monades. Considérons une définition de lift1 (qui est légèrement différente de liftM1 de la bibliothèque Monad) :

```
lift1   :: (a -> b) -> (R a -> R b)
lift1 f =  \ra1 -> do a1 <- ra1
                      step (f a1)
```

Cela prend une fonction à un argument, f, et crée une fonction dans R qui exécute la fonction liftée en une étape. En utilisant lift1, inc est devenue :

```
inc   :: R Integer -> R Integer
inc i =  lift1 (i+1)
```

Cela est mieux, mais pas encore idéal. Dans un premier temps, on va ajouter lift2 :

```
lift2    :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f  =  \ra1 ra2 -> do a1 <- ra1
                           a2 <- ra2
                           step (f a1 a2)
```

Remarquez que la fonction définit explicitement l'ordre d'évaluation de la fonction liftée : l'exécution concernant a1 se produira avant celle pour a2.
En utilisant lift2, on peut créer une nouvelle version de == dans la monade R :

```
(==*) : : Ord un => R a - > R a - > R Bool
(==*) = lift2 (==)
```

On a dû employer un nom légèrement différent pour cette nouvelle fonction, puisque le == est déjà pris. Mais dans certains cas, on peut utiliser le même nom pour la fonction liftée et pour celle d'origine. Cette déclaration d'instance permet à tous les opérateurs de Num d'être utilisées dans R:

```
instance Num a => Num (R a) where
  (+)                    =  lift2 (+)
  (-)                    =  lift2 (-)
  negate                 =  lift1 negate
  (*)                    =  lift2 (*)
  abs                    =  lift1 abs
  fromInteger            =  return . fromInteger
```

La fonction fromInteger est appliquée implicitement à toutes les constantes entières dans Haskell (voir la section X-3). La définition permet aux constantes entières d'avoir le type R Integer. On peut maintenant, enfin, écrire une fonction d'incrémentation dans un style complètement naturel :

```
inc                     :: R Integer -> R Integer
inc x                   =  x + 1
```

Remarquez qu'on ne peut pas lifter la classe Eq de la même manière que la classe Num. En effet, la signature de ==* n'est pas compatible avec les surcharges autorisées de ==, car le résultat de ==* est de type R Bool et non pas Bool.

Pour exprimer des exécutions intéressantes dans R, on va avoir besoin de conditions. Comme on ne pourra pas utiliser le nom if, qui requiert que le test soit de type Bool et non R Bool, on prendra le nom ifR.

```
ifR                     :: R Bool -> R a -> R a -> R a
ifR tst thn els         =  do t <- tst
                              if t then thn else els
```

Maintenant, on est prêt pour un long programme dans la monade R.

```
fact                    :: R Integer -> R Integer
fact x                  =  ifR (x ==* 0) 1 (x * fact (x-1))
```

Désormais, ce sera bien moins facile qu'avec la simple fonction factorielle, mais ça devrait tout de même rester lisible. L'idée de proposer des nouvelles définitions pour les opérations existantes comme + ou if est une partie essentielle de la création du langage embarqué dans Haskell. Les monades sont particulièrement utiles pour encapsuler les sémantiques de ces langages de manière propre et modulaire.

On est maintenant prêt pour faire tourner quelques programmes. Cette fonction lance un programme dans M en lui donnant le nombre maximal d'étapes d'exécution.

```
run                     :: Resource -> R a -> Maybe a
run s (R p)             =  case (p s) of
                             (_, Left v) -> Just v
                             _           -> Nothing
```

On peut utiliser le type Maybe pour donner à l'exécution la possibilité de ne pas finir dans

le nombre d'étapes accordé. Ainsi, on peut obtenir :

run 10 (fact 2) => Just 2
run 10 (fact 20) => Nothing

Finalement, on peut ajouter des fonctionnalités intéressantes à cette monade. Regardons la fonction suivante :

```
(|||)                   :: R a -> R a -> R a
```

Cela lance en parallèle deux exécutions, et renvoie la valeur de la première. Une définition possible d'une telle fonction serait :

```
c1 ||| c2              =  oneStep c1 (\c1' -> c2 ||| c1')
   where
        oneStep        :: R a -> (R a -> R a) -> R a
        oneStep (R c1) f =
             R (\r -> case c1 1 of
                        (r', Left v) -> (r+r'-1, Left v)
                        (r', Right c1') ->  -- r' must be 0
                         let R next = f c1' in
                           next (r+r'-1))
```

Cette fonction prend une étape dans c1, et renvoie sa valeur complète c1, ou si c1 a renvoyé une exécution suspendue (c1'), évalue c2 ||| c1'. La fonction oneStep prend une étape élémentaire en argument, et renvoie la valeur évaluée, ou passe le reste de l'exécution à f. La définition de oneStep est simple: elle donne à c1 un 1 comme valeur d'entrée. Si la valeur finale est atteinte, elle est renvoyée, ce qui ajuste le compteur d'étapes renvoyé (il est possible que l'exécution renvoit, après n'avoir pris aucune étape, donc le compteur de ressources renvoyé n'est pas forcemment 0). Si l'exécution est suspendue, un compteur de ressource "patché" sera passé à la continuation finale.

On peut maintenant évaluer des expressions comme run 100 (fact (-1) ||| (fact 3)) sans boucler grâce à l'exécution de deux processus en parallèle. (Notre définition de fact boucle pour -1). De nombreuses variantes sont possibles autour de cette structure de base. Par exemple, on peut étendre l'état, de façon à inclure la trace de toutes les étapes de l'exécution. On peut aussi inclure cette monade dans la monade standard IO, ce qui autoriserait les exécutions dans M à interagir avec le reste du monde.
Même si cet exemple est peut-être plus avancé que les autres de ce tutoriel, il sert à illustrer la puissance des monades, en tant qu'outils pour définir la sémantique de base d'un système. On présente aussi cet exemple comme un modèle d'un petit langage spécifique à un domaine (DSL), domaine que Haskell peut particulièrement bien définir. De nombreux autres DSLs ont été développés dans Haskell; regardez haskell.org pour plus de détails. Les plus intéressants sont Fran, un langage d'animations réactives, et Haskore, un langage de synthèse musicale.

X. Les Nombres
X-1. Structure des classes numériques
X-2. Les Nombres construits
X-3. Les Conversions numériques et les surcharges de littéraux
X-4. Les Types numériques par défaut

X. Les Nombres

Haskell fournit une collection très complète de types numériques, basée sur le modèle de Scheme, qui découlent eux-mêmes de ceux de Common Lisp. Toutefois, ces langages sont typés dynamiquement. Les types standards contiennent déjà des entiers à précision fixé ou arbitraire, des nombres rationnels formés à partir des types entiers, des chiffres réels à simple ou double précision, et des nombres complexes à "virgule flottante". On ne montrera que les caractéristiques de base de la structure des classes de types numériques, et le lecteur est invité à se référer à la section 6.4 pour de plus amples détails.

X-1. Structure des classes numériques

Les classes des types numériques (Num et celles qui en découlent) forment une grande partie des classes de la librairie standard d'Haskell. On remarquera que Num est une classe de Eq, mais pas de Ord. En effet, la notion d'ordre ne s'applique pas aux nombres complexes. Cependant, la sous-classe Real héritée de Num est bien une sous-classe de Ord.

La classe Num fournit plusieurs opérations de base communes à tous les types numériques. On y trouve, entre autres, l'addition, la soustraction, la négation, la multiplication, et la valeur absolue :

```
(+), (-), (*)              :: (Num a) => a -> a -> a
negate, abs                :: (Num a) => a -> a
```

La négation est une fonction appliquée par le seul opérateur préfixe de Haskell, -, on ne peut pas l'appeler (-), car cela représente la fonction soustraction. On a donc été obligé de lui donner un nom à la place. Par exemple, -x*y est équivalent à negate (x*y). Notez que l'opérateur préfixe moins a la même priorité que l'opérateur infixe moins, qui est plus faible que celle de la multiplication.

Remarquez que Num ne fournit pas d'opération division. En effet, il y existe deux types de division qui sont fournis par des types non disjoints :

* La classe Integral fournit la division entière et le modulo. Les instances standards de Integral sont Integer, entiers au sens mathématique donc non bornés (aussi appelés "bignums"), et Int, représentant des entiers machine bornés codés sur au moins 29 bits signés. Une implémentation particulière de Haskell peut fournir d'autres types entiers en supplément. Remarquez que Integral est une sous-classe de Real, et non une sous-classe

directe de Num. Cela signifie que l'on n'essaie pas de fournir des entiers de Gauss.

* Les autres types numériques découlent de Fractional, qui fournit une opération de division classique, (/). Sa sous-classe Floating contient des fonctions trigonométriques, logarithmiques et exponentielles.

* La sous-classe RealFrac héritant de Fractional et de Real fournit une fonction properFraction, qui décompose le nombre en sa partie entière et sa partie fractionnaire, et un ensemble de fonctions qui arrondissent les valeurs selon différentes règles :

```
properFraction  :: (Fractional a, Integral b) => a -> (b,a)
truncate, round,
floor, ceiling: :: (Fractional a, Integral b) => a -> b
```

* La sous-classe RealFloat héritant de Floating et de RealFrac fournit quelques fonctions spécialisées pour accéder efficacement aux composantes des nombres à virgules, nommées exponent et significand.

* Les types standards Float et Double dérivent de la classe RealFloat.

X-2. Les Nombres construits

Parmi les types numériques standards, Int, Integer, Float et Double sont primitifs. Les autres sont construits à partir de ceux-ci, grâce à des constructeurs de type.

Complex, située dans la librairie Complex, est un constructeur de type qui produit un type complexe de la classe Floating à partir du type RealFloat :

```
data (RealFloat a) => Complex a = !a :+ !a  deriving (Eq, Text)
```

Les symboles ! sont des marqueurs de restriction, dont on a parlé dans la section 6.3. Remarquez que le contexte RealFloat a restreint le type de l'argument. Par conséquent, les types complexes standards sont Complex Float et Complex Double. On peut aussi voir dans la déclaration de données que le nombre complexe est écrit x :+ y, les arguments sont les parties réelles et imaginaires. Comme :+ est un constructeur de donnée, on peut l'utiliser dans la reconnaissance de motif :

```
conjugate         :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y) =  x :+ (-y)
```

De la même manière, le constructeur de type Ratio, situé dans la librairie Rational, produit un type rationnel de classe RealFrac à partir d'une instance de type Integral. Au passage, il faut signaler que Rational est synonyme de Rational Integer. Cependant, Ratio est un constructeur de type abstrait. Au lieu d'utiliser un constructeur de données comme :+, les rationnels utilisent la fonction % pour créer un ratio à partir de deux entiers. Au lieu d'utiliser de la reconnaissance de motif, des fonctions d'extraction des composantes sont fournies :

```
(%)                        :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
```

Pourquoi a-t-on fait une différence ? Les nombres complexes sous forme cartésienne sont uniques, et ne font pas d'égalités non triviales impliquant :+. D'autre part, les ratios ne sont pas uniques, mais possèdent une forme réduite que l'implantation du type abstrait doit conserver. Il n'est pas certain, par exemple, que numerator (x%y) soit égal à x, bien que la partie réelle de x:+y soit toujours x.

X-3. Les Conversions numériques et les surcharges de littéraux

Le Prélude standard et ses librairies fournissent plusieurs fonctions surchargées qui servent aux conversions explicites :

```
fromInteger            :: (Num a) => Integer -> a
fromRational           :: (Fractional a) => Rational -> a
toInteger              :: (Integral a) => a -> Integer
toRational             :: (RealFrac a) => a -> Rational
fromIntegral           :: (Integral a, Num b) => a -> b
fromRealFrac           :: (RealFrac a, Fractional b) => a -> b

fromIntegral           =  fromInteger . toInteger
fromRealFrac           =  fromRational . toRational
```

Deux d'entre elles sont implicitement utilisées pour produire des littéraux numériques surchargés : Un entier numérique (sans décimales) est effectivement équivalent à l'application de fromInteger à la valeur d'un nombre de type Integer. De la même manière, un nombre décimal est vu comme une application de fromRational à la valeur d'un nombre de type Rational. Par conséquent, 7 a le type (Num a) => a, et 7.3 a le type (Fractional a) => a. Cela signifie que l'on peut choisir d'utiliser des littéraux numériques dans des fonctions numériques génériques. Par exemple :

```
halve   :: (Fractional a) => a -> a
halve x =  x * 0.5
```

Cette manière indirecte de surcharger des littéraux numériques a un avantage supplémentaire : la méthode qui interprétait un littéral numérique comme un nombre d'un type donné peut être spécifié dans une déclaration d'instance de la classe Integral ou Fractional, grâce aux fonctions fromInteger et fromRational. Par exemple, l'instance Num de (RealFloat a) => Complex a contient cette méthode :

```
fromInteger x =  fromInteger x :+ 0
```

Cela dit que l'instance Complex de fromInteger est définie pour produire un nombre complexe dont la partie réelle est fournie par une instance RealFloat appropriée de fromInteger. De cette manière, même les types numériques définis par l'utilisateur (ex: les quaternions) peuvent utiliser des surcharges.

Si vous souhaitez un autre exemple, rappelez-vous de notre première définition de inc à la

section 2 :

```
inc    :: Integer -> Integer
inc n  =  n+1
```

Si l'on ignore la signature du type, le type le plus général pour inc est (Num a) => a->a. Cependant, la signature explicite du type est valide parce que elle est plus spécifique que le type principal. Une signature de type plus générale aurait provoqué une erreur de typage statique. La signature de type a pour effet de restreindre le type de inc, et dans ce cas aurait causé quelque chose comme inc (1::Float), qui est mal typé.

X-4. Les Types numériques par défaut

Considérez la définition de fonction suivante :

```
rms     :: (Floating a) => a -> a -> a
rms x y =  sqrt ((x^2 + y^2) * 0.5)
```

La fonction d'exponentiation (^), qui est l'une des trois opérations dont les types sont différents fournies en standard pour l'exponentiation (cf section 6.8.5), a le type (Num a, Integral b) => a -> b -> a. Par ailleurs, étant donné que 2 a le type (Num a) => a, le type de x^2 est (Num a, Integral b) => a. Ceci pose un problème, car il n'y a pas moyen de résoudre la surcharge associée à une variable de type b. En effet, bien que elle soit dans le contexte, l'information a disparu du type de l'expression. Pour faire simple, le programmeur a spécifié que x devrait être élevé au carré, mais n'a pas précisé si le type de 2 était Int ou Integer. Bien entendu, cela peut se corriger :

```
rms x y =  sqrt ((x ^ (2::Integer) + y ^ (2::Integer)) * 0.5)
```

Cependant, il est évident que de tels événements réapparaitront tôt ou tard.

En fait, ce genre d'ambiguïté sur la surcharge ne touche pas que les nombres :

```
show (read "xyz")
```

Sous quel type est-on censé lire la chaîne de caractères ? Ce problème peut être plus sérieux qu'avec l'exponentiation, car là-bas n'importe quelle instance de type Integral fonctionnait, alors qu'ici différents comportements peuvent être souhaité selon l'instance de Text que l'on utilisera pour lever l'ambiguïté.

En raison de la différence entre les cas numérique et quelconque des problèmes d'ambiguïté lors de la surcharge, Haskell fournit une solution qui ne s'applique qu'aux nombres. Chaque module peut contenir une déclaration "par défaut" suivie par une liste de types numériques simples (sans variables) entre parenthèses et séparés par des virgules. Quand une ambiguïté sur le type d'une variable est découverte, comme avec b, si au moins l'une des classes est numérique et que toutes ses classes sont standards, la liste

par défaut sera consultée, et le premier type valide trouvé sera utilisé. Par exemple, si la déclaration par défaut est default (Int, Float), une exponentiation ambigüe sera résolue avec le type Int. (voir la section 4.3.4 pour plus de détails)

Par défaut, on utilise la liste par défaut (Integer, Double), mais (Integer, Rational, Double) peut aussi convenir. Les programmeurs très prudents peuvent préférer default (), ce qui supprime les choix par défaut.

XI. Les Modules
XI-1. Les Noms qualifiés
XI-2. Les Types de données abstraits
XI-3. Plus de caractéristiques

XI. Les Modules

Un programme Haskell est constitué d'un ensemble de modules. En Haskell, un module a une double utilité : il sert à contrôler les espaces de nommage, et à créer un type de données abstrait.

Le niveau supérieur d'un module contient toutes les déclarations dont nous avons parlé jusqu'ici : déclarations de fixité, déclarations de type et de données, déclarations de classes et d'instances, signatures de types, définitions de fonctions, et liens avec des motifs. Excepté le fait que les déclarations d'importation (que nous déjà décrites brièvement) doivent apparaître au début, les déclarations peuvent apparaître dans n'importe quel ordre, la portée du niveau supérieur étant mutuellement récursive.

La conception de modules en Haskell est relativement conservatrice : l'espace de nommage des modules est complètement plat, et les modules ne sont aucunement des classes. Les noms des modules sont alphanumériques et doivent commencer par une lettre en majuscule. Il n'y a pas de liens directs entre un module Haskell et le fichier système qui le contient. En particulier, il n'y a aucune correspondance entre les noms de modules et les noms des fichiers, plus d'un module peuvent être stockés dans un seul fichier, ou un module peut résider dans plusieurs fichiers. Bien entendu, une implantation particulière adoptera certainement des conventions reliant modules et fichiers de manière plus contraignante.

Techniquement parlant, un module n'est qu'une grosse déclaration qui commence avec le mot-clé module. Voici l'exemple d'un module nommé Tree :

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Le type Tree et la fonction fringe devraient vous être familières, car ils ont été donnés en exemple dans la section 2.2.1. En raison du mot-clé where, la mise en forme est active dès le début du module, et par conséquent les déclarations doivent toutes être alignées sur la même colonne (typiquement la première). Ainsi on remarque que le nom du module est le même que celui du type, ce qui est autorisé.

Le module exporte explicitement Tree, Leaf, Branch et fringe. Si la liste des exportations

suivant le mot-clé module est omis, tous les noms liés au niveau supérieur du module seront exportés. Dans l'exemple donné, tout est explicitement exporté, ce qui a le même effet. Remarquez que le nom du type et de son constructeur ont été groupés ensemble, comme dans Tree(Leaf,Branch). On peut aussi utiliser un raccourci en écrivant Tree(...). Exporter un sous-ensemble de constructeurs est ainsi possible. Les noms dans la liste des exportations doivent ne pas être locaux au module exporté. N'importe quel nom dans la portée peut être placé dans la liste des exportations.

Le module Tree peut désormais être importé dans un autre module :

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

Les divers éléments qui vont être importés dans le module et exportés hors de celui-ci sont appelés des entités. Remarquez la liste des importations explicite dans la liste des déclarations, sans quoi toutes les entités exportées de Tree auraient été importées.

XI-1. Les Noms qualifiés

Il y a un problème évident dans l'importation des noms directement dans l'espace de nommage du module. Que se passerait-il si deux modules importés contenaient des entités différentes ayant le même nom ? Une déclaration d'importation peut utiliser le mot-clé qualifié pour préfixer les noms par le nom du module dans lequel ils sont. Ces préfixes sont suivis par le caractère . sans espace intercalés. Les qualifieurs sont partis de la syntaxe lexicale. Par conséquent, A.x et A . x sont totalement différents : le premier est un nom qualifié et le second est l'utilisation de la fonction infixe . Par exemple, en utilisant le module Tree introduit précédemment :

```
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]   -- A different definition of fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x

module Main where
import Tree ( Tree(Leaf,Branch), fringe )
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Quelques programmeurs Haskell préfèrent utiliser les qualificateurs pour toutes les entités importées, en rendant explicite l'origine de chaque nom à chaque utilisation. D'autres préfèrent les noms courts et utilisent uniquement des qualificateurs lorsque cela s'avère nécessaire.

Les qualificateurs sont utilisés pour résoudre les conflits entre différentes entités qui ont le même nom. Mais que se passerait-il si la même entité était importée depuis plus d'un module ? Heureusement, de tels conflits de noms sont autorisés : une entité peut être importée par différents chemins sans provoquer de conflit. Le compilateur sait si les entités de différents modules sont effectivement identiques.

XI-2. Les Types de données abstraits

En plus de définir des espaces de nommage, les modules fournissent la seule méthode de construire des types de données abstraits en Haskell. Par exemple, une caractéristique d'un ADT est que le type représenté est caché. Toutes les opérations sur cet ADT sont faites à un niveau abstrait qui ne dépend pas de cette représentation. Ainsi, bien que le type Tree soit assez simple pour que nous n'ayons pas besoin de recourir à une abstraction, un ADT adapté pourrait inclure les opérations suivantes :

```
data Tree a             -- just the type name
leaf                    :: a -> Tree a
branch                  :: Tree a -> Tree a -> Tree a
cell                    :: Tree a -> a
left, right             :: Tree a -> Tree a
isLeaf                  :: Tree a -> Bool
```

Un module supportant cela peut être par exemple celui-ci :

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

data Tree a             = Leaf a | Branch (Tree a) (Tree a)

leaf                    = Leaf
branch                  = Branch
cell  (Leaf a)          = a
left  (Branch l r)      = l
right (Branch l r)      = r
isLeaf   (Leaf _)       = True
isLeaf   _              = False
```

Remarquez que dans la liste des exportations, le nom du type Tree apparaît seul (ie sans constructeurs). Par conséquent Leaf et Branch ne seront pas exportés, et le seul moyen de construire ou d'extraire une partie d'un arbre de ce module est d'utiliser différents opérations abstraites. Bien entendu, le bénéfice qu'on tire de cacher ces informations est que, plus tard, nous pouvons changer le type dans lequel est stockée la donnée sans affecter les utilisateurs de ce type.

XI-3. Plus de caractéristiques

Vous trouverez ici un bref aperçu de quelques autres caractéristiques du système de modules. Vous pouvez vous reporter au rapport pour plus de détails...

* Une déclaration d'importation peut cacher des entités sur demande en utilisant la clause hiding dans la déclaration. Cela peut s'avérer utile pour exclure explicitement des noms qui sont utilisés pour d'autres objectifs sans avoir à utiliser des qualificateurs pour les noms importés depuis d'autres modules.

* Une importation peut contenir une clause as pour spécifier un qualificateur différent que le nom du module importé. Cela permet de raccourcir les noms de modules très longs ou d'adapter facilement le code lors d'un changement de module sans avoir à modifier tous les qualificateurs.

* Les programmes importent implicitement le module Prelude. Une importation explicite de ce module écraserait les noms importés lors de l'importation implicite. Par conséquent, le code ci-dessous n'importera pas length depuis le Prélude standard, ce qui permettra de définir ce nom différemment.

```
import Prelude hiding length
```

* Les déclarations d'instances ne sont pas explicitement nommées dans les listes d'importation ou d'exportation. Chaque module exporte toutes ses déclarations d'instances et chaque importation apporte toutes les déclarations d'instance dans sa portée.

* Les méthodes des classes peuvent être nommées soit de la manière des constructeurs de données, dans les parenthèses suivant le nom de la classe, soit comme des variables ordinaires.

Bien que le système de modules d'Haskell soit relativement conservateur, il y a de nombreuses règles concernant l'importation et l'exportation des valeurs. La majorité d'entre elles sont évidentes. Par exemple, il est interdit d'importer deux entités différentes ayant le même nom dans la même portée. D'autres règles le sont moins. Par exemple, pour un type et une classe donnés, il ne peut pas y avoir plus d'une déclaration d'instance pour cette combinaison (type,classe) ailleurs dans le programme.
Le lecteur devrait lire le rapport en détails (section 5).

XII. Pièges du typage
XII-1. Let-Bound Polymorphism
XII-2. Surcharge numérique
XII-3. Les Restrictions monomorphiques

XII. Pièges du typage

Cette courte partie devrait vous donner une bonne intuition en ce qui concerne les problèmes que le système de type de Haskell pose aux débutants.

XII-1. Let-Bound Polymorphism

Tout langage utilisant le système de type de Hindley-Milner possède ce que l'on appelle let-bound polymorphism, parce que les identifiants non bornés par une clause let ou where, et non situés au niveau principal d'un module, sont limités par leur propre polymorphisme. En particulier, une fonction lambda-bound, c'est-à-dire qui est passée en argument à une autre fonction, ne peut pas être instanciée de deux manières différentes. Par exemple, le programme suivant est invalide :

```
let f g  =  (g [], g 'a')    -- ill-typed expression
in f (\x->x)
```

En effet, le représentant d'une fonction abstraite, dont le type principal est a->a, est utilisée dans f de deux manières différentes : la première fois avec le type [a]->[a] ; et la seconde avec le type Char->Char.

XII-2. Surcharge numérique

Il est facile d'oublier que, parfois, les valeurs numériques sont surchargées, et pas implicitement converties dans les divers types numériques comme dans beaucoup d'autres langages. Ainsi des expressions numériques très générales ne peuvent parfois pas être généralisées. On rencontre souvent une erreur de typage pour des valeurs numériques comme celle-ci :

```
xs moyens = xs de somme/xs de longueur           -- Mal !
```

/ exige des arguments de type fraction, mais le résultat de length est de type Int. Cette erreur de type doit être corrigée en rendant le transtypage explicite :

```
average                 :: (Fractional a) => [a] -> a
average xs              =  sum xs / fromIntegral (length xs)
```

XII-3. Les Restrictions monomorphiques

Le système de types de Haskell contient une restriction liée aux classes de type qui n'est pas présente dans le système de types standard de Hindley-Milner : la restriction de

monomorphisme. La raison de cette restriction provient d'une subtile ambiguïté de types et est entièrement expliqué dans ce rapport. Pour faire court, on peut le résumer comme :

La restriction de monomorphisme indique que n'importe quel indentifiant borné par un binding (cela inclut les bindings sur les identifiants seuls), et n'ayant aucun type explicite dans sa signature, doit être monomorphique. Un identifiant est monomorphique s'il n'est pas surchargé, ou bien s'il est surchargé mais est employé de manière à n'obtenir au plus qu'une seule surcharge et n'est pas exporté.

Les violations de cette restriction ont comme conséquence une erreurde typage statique. La manière la plus simple d'éviter ce problème est de fournir un type explicite dans la signature. On peut remarquer que n'importe quel type mis en signature suffira, du moment qu'il s'agit d'un type correct.

Une violation courante de cette restriction survient lorsqu'on définit des fonctions d'ordre supérieur. En voici un exemple avec la définition classique de la somme :

```
sum                    =  foldl (+) 0
```

On peut corriger cette erreur de typage statique, en ajoutant un type dans la signature, comme suit :

```
sum                    :: (Num a) => [a] -> a
```

On remarque ainsi que le problème ne serait pas survenu si nous avions écrit ceci :

```
sum xs                 =  foldl (+) 0 xs
```

En effet, la restriction n'est appliquée que dans les binding.

XIII. Les Tableaux
XIII-1. Les types d'indice
XIII-2. Création d'un tableau
XIII-3. Accumulation
XIII-4. Mises à jour incrémentales
XIII-5. Un exemple : la multiplication matricielle

XIII. Les Tableaux

Idéalement en programmation fonctionnelle, les tableaux devraient êtres vus comme des fonctions associant à un entier un élément du tableau. Mais si l'on souhaite être plus pragmatique, et donc privilégier les performances des accès aux éléments, on doit assurer quelques propriétés sur le domaine de ces fonctions, de façon à garder l'isomorphisme avec les sous-ensembles contigus d'entiers. Par conséquent, Haskell ne traitera pas les tableaux comme des fonctions, au sens le plus général du terme, mais plutôt comme un type de données abstrait avec une opération associée.

Il existe en gros deux grandes approches dans la gestion fonctionnelle des tableaux : la définition incrémentale, et la monolithique.
Dans l'approche incrémentale, on a une fonction renvoyant un tableau vide d'une taille donnée, et une autre qui prend un tableau, un index et une valeur, et qui renvoie un nouveau tableau qui diffère de celui passé en argument par la valeur au niveau de l'index donné en argument. Evidemment, une implantation naïve d'une telle méthode donnerait des performances désastreuses, car elle nécessite la recopie du tableau entier à chaque modification d'un élément (modification incrémentale) tout en donnant un temps de parcours linéaire. Par conséquent, toutes les tentatives sérieuses utilisent une approche employant une analyse statique fine, et des techniques rusées afin d'éviter les recopies excessives.
Dans l'approche monolithique, on construit le tableau complet en une fois, sans garder les valeurs intermédiaires du tableau. Bien que Haskell utilise un opérateur de mise à jour incrémentale pour les tableaux, la plupart de sa manipulation de tableaux est d'inspiration monolithique.

Les tableaux ne font pas partie du prélude standard, bien que la librairie standard intègre les opérateurs sur les tableaux. Il faut donc charger le module Array pour pouvoir les utiliser.

XIII-1. Les types d'indice

La librairie Ix définit une classe de type pour les indices de tableaux :

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

Les déclarations d'instances sont effectuées pour les types Int, Integer, Char, Bool et les nuplets de Ix dont la longueur est inférieure ou égale à 5. Par ailleurs, les instances peuvent être automatiquement dérivées pour les types nuplets ou énumérés. On considère les types primitifs comme des indices de tableaux, et les nuplets comme des indices de tableaux multi-dimensionnels. Remarquez que le premier argument de chaque opération de la classe Ix est une paire d'indices, qui sont les bornes (premier et dernier indices) du tableau. Par exemple, les bornes d'un tableau de 10 éléments d'un type Int, dont l'origine est nulle, seront (0,9), alors que celles d'un tableau de taille 100x100, dont l'origine est 1, seront ((1,1),(100,100)).
(Dans de nombreux autres langages, de telles bornes seraient écrites sous la forme 1:100, 1:100, mais la forme utilisée ici convient mieux au système de type, car chaque borne possède le même type que l'index)

L'opération range prend en argument une paire de bornes, et renvoie la liste des indices contenus entre ces bornes, dans l'ordre de l'index. Voici un exemple :

```
range (0,4) => [0,1,2,3,4]

range ((0,0),(1,2)) => [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]
```

Le prédicat inRange détermine si un index est bien situé entre une paire de bornes donnée. (Pour un type nuplet, ce test est effectué composante par composante).
Morceau à corriger Enfin, l'opération index autorise un élément particulier du tableau à être adressé directement. Etant donné une paire de bornes et un index in-range, l'opération va aligner l'ordinal de l'origine de l'index. Voici un exemple :

```
index (1,9) 2 => 1

index ((0,0),(1,2)) (1,1) => 4
```

XIII-2. Création d'un tableau
Dans Haskell, la fonction de création d'un tableau par la méthode monolithique renvoie un tableau à partir d'une paire de bornes et d'une liste de couple (index,valeur) (une liste associative)

```
array          :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Ici, par exemple, on définit un tableau contenant les carrés des entiers de 1 à 100 :

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

Cette expression d'un tableau est caractéristique dans l'utilisation de la compréhension qu'on a des listes pour construire des listes associatives. En fait, cette utilisation permet une compréhension des expressions de tableau similaires aux array comprehension du

langage Id.

L'accès direct aux éléments d'un tableau est effectué par l'opérateur infixe !, et les bornes du tableau peuvent être obtenues avec la fonction bounds :

```
squares!7 => 49

bounds squares => (1,100)
```

On peut généraliser cet exemple en paramétrisant les bornes et la fonction qui doit être appliquée à chaque index :

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

Par conséquent, on peut définir squares comme mkArray (\i -> i * i) (1,100)

```
fibs    :: Int -> Array Int Int
fibs n  =  a  where
               a = array (0,n)
                         ([(0, 1), (1, 1)]
                           ++
                          [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

Un autre exemple d'une telle récurrence est la matrice wavefront, dans laquelle les éléments de la première ligne et de la première colonne valent 1, et les autres valent la somme des éléments situés au nord, au nord-ouest et à l'ouest :

```
wavefront   :: Int -> Array (Int,Int) Int
wavefront n =  a  where
                 a = array ((1,1),(n,n))
                     ([((1,j), 1) | j <- [1..n]] ++
                      [((i,1), 1) | i <- [2..n]] ++
                      [((i,j),a!(i,j-1)+a!(i-1,j-1)+a!(i-1,j))
                                   | i <- [2..n], j <- [2..n]])
```

On l'a appelée matrice wavefront car dans une exécution en parallèle, la récurrence implique que l'exécution peut commencer avec la première ligne et la première colonne en parallèle, puis doit progresser comme une onde traversant la matrice du nord-ouest au sud-est. Il est important de signaler que, cependant, l'ordre de l'exécution n'est pas spécifié par la liste associative.

Dans chacun de nos exemples, on a donné une unique association pour chaque index du tableau, et seulement pour des indices situés entre les bornes du tableau. En effet, on doit faire cela de manière générale pour que le tableau soit complètement défini.
Une association avec un index dépassant les bornes produira une erreur. Si l'index manque ou apparait plus d'une fois, il n'y aura cependant pas d'erreur immédiatement, mais la valeur du tableau à cet index sera non-définie, et son accès produira une erreur.

XIII-3. Accumulation
On peut diminuer les contraintes lorsqu'un index apparait au plus une fois dans la liste associative, en spécifiant comment combiner les valeurs multiples associées à un seul index. Le résultat est appelé un tableau accumulé.

accumArray :: (Ix a) -> (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b

Le premier argument de accumArray est la fonction d'accumulation, le second est la valeur initiale (la même pour chaque élément du tableau), et les arguments restants sont les bornes et la liste associative, comme avec la fonction array.
Un exemple typique serait d'avoir la fonction d'accumulation (+), et une valeur initiale 0. Ainsi, cette fonction prend en arguments une paire de bornes et une liste de valeurs (du type de l'index), et renvoie un histogramme, c'est-à-dire une table contenant le nombre d'occurrences de chaque valeur entre les bornes.

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <- is, inRange bnds i]
```

Supposons qu'on ait un ensemble de mesures situés dans un intervalle [a,b], et que l'on veuille diviser l'intervalle en dizaines, et compter le nombre de mesures présentes dans chaque sous-intervalle :

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade
                   where decade x = floor ((x - a) * s)
                         s        = 10 / (b - a)
```

XIII-4. Mises à jour incrémentales
En plus de ses fonctions de création de tableaux de manière monolithique, Haskell dispose aussi d'une fonction de mise à jour incrémentale pour les tableaux, écrite par l'opérateur infixe //. Le cas le plus simple est un tableau a ayant un élément i qu'il doit mettre à jour avec la valeur v. Cela s'écrit a // [(i, v)]. On doit utiliser un encadrement entre crochets car l'argument gauche de (//) est une liste associative, qui contient généralement un ensemble d'indices du tableau qui lui est propre.

(//)   :: (Ix a) => Array a b -> [(a,b)] -> Array a b

Tout comme la fonction array, les indices d'une liste associative doivent être uniques pour que les valeurs associées puissent être définies. Par exemple, considérons une fonction qui intervertit deux lignes d'une matrice :

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a =  a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                         [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

La concaténation de deux listes séparées dans une liste d'indices j n'est cependant pas vraiment efficace. Cela revient à écrire deux boucles imbriquées comme l'on ferait avec un langage impératif. Mais n'ayez crainte, on peut effectuer une optimisation proche de la fusion des boucles en Haskell :

```
swapRows i i' a =  a // [assoc | j <- [jLo..jHi],
                                 assoc <- [((i ,j), a!(i',j)),
                                           ((i',j), a!(i, j))] ]
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

XIII-5. Un exemple : la multiplication matricielle

On peut compléter cette introduction aux tableaux en Haskell avec l'exemple typique de la multiplication matricielle, en se servant d'une surcharge d'une fonction bien plus générale. En effet, seules la multiplication et l'addition pour le type des éléments de la matrice sont nécessaires, on peut donc facilement écrire une fonction qui effectue la multiplication de matrices quelconques sans que cela exige beaucoup plus d'effort. Par ailleurs, si l'on fait attention à n'appliquer (!) et les opérations de Ixaux indices, on peut gagner en généricité sur les types des index, et en fait les types des index de lignes et de colonnes n'auront pas besoin d'être identiques. Par soucis de simplicité, on demandera cependant que les indices à gauche des colonnes, et à droite des lignes soient de même type, et en plus que leurs bornes soient égales.

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                   Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =  array resultBounds
                         [((i,j), sum [x!(i,k) * y!(k,j) | k <- range (lj,uj)])
                                      | i <- range (li,ui),
                                        j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

A côté de cela, on peut aussi définir matMult utilisant accumArray, et qui renverra le résultat sous une forme qui ressemble plus à la forme usuelle d'une langage impératif :

```
matMult x y     =  accumArray (+) 0 resultBounds
                              [((i,j), x!(i,k) * y!(k,j))
                                      | i <- range (li,ui),
                                        j <- range (lj',uj')
                                        k <- range (lj,uj)  ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

On peut généraliser encore plus cette fonction en passant à l'ordre supérieur, juste en remplaçant sum et (*) par des fonctions passées en paramètres :

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult sum' star x y  =
      array resultBounds
            [((i,j), sum' [x!(i,k) `star` y!(k,j) | k <- range (lj,uj)])
                                 | i <- range (li,ui),
                                   j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

Les fans d'APL reconnaîtront l'utilité des fonctions suivantes :

```
genMatMult maximum (-)
genMatMult and (==)
```

Avec la première, les arguments sont des matrices numériques, et l'élément (i,j) du résultat est le maximum de la différence entre les éléments correspondants à la ligne i du premier argument et la colonne j du second.
Avec la seconde, les arguments sont des matrices de type quelconque, et le résultat sera une matrice booléenne où chaque élément (i,j) est vrai si et seulement si la ligne i du premier argument et la colonne j du second sont égaux en tant que vecteurs.

Remarquez que les éléments de genMatMult n'ont pas besoin d'avoir des types identiques, mais seulement compatibles avec l'opérateur star passé en argument. On peut d'ailleurs encore généraliser en oubliant l'obligation que le type des lignes du premier argument soit identique à celui des colonnes du second. En effet, les deux matrices pourront être valides tant que la longueur des lignes du premier est égale à celle des colonnes du second.
Le lecteur peut souhaiter en dériver une version encore plus générale (indication: Utilisez l'opération index pour déterminer les longueurs)

XIV. Conclusion
XIV-1. Prochaine étape
XIV-2. Remerciements

XIV. Conclusion

XIV-1. Prochaine étape

On peut trouver beaucoup de ressources Haskell sur haskell.org. Vous y trouverez des compilateurs, des démos, des articles et beaucoup d'informations intéressantes à propos de Haskell et des langages fonctionnels. Les compilateurs et les interpréteurs Haskell fonctionnement sur quasiment tout type de matériel et de système d'exploitation. Le système Hugs est léger et portable, c'est un excellent outil pour apprendre Haskell.

XIV-2. Remerciements

Je remercie Patricia Fasel et Mark Mundt à Los Alamos, et Nick Carriero, Charles Consel, Amir Kishon, Sandra Loosemore, Martin Odersky, et David Rochberg de l'université de Yale pour leurs relectures rapides des premières versions de cet article. Je remercie tout particulièrement Erik Meijer pour ses commentaires approfondis sur les ajouts de la version 1.4 de ce tutoriel.

Bibliographie

[1] R. Bird. Introduction to Functional Programming using Haskell. Prentice Hall, New York, 1998.

[2] A.Davie. Introduction to Functional Programming System Using Haskell. Cambridge University Press, 1992.

[3] P. Hudak. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, 21(3):359--411, 1989.

[4] Simon Peyton Jones (editor). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106, Feb 1999.

[5] Simon Peyton Jones (editor) The Haskell 98 Library Report. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1105, Feb 1999.

[6] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[7] J. Rees and W. Clinger (eds.). The revised3 report on the algorithmic language Scheme. SIGPLAN Notices, 21(12):37--79, December 1986.

[8] G.L. Steele Jr. Common Lisp: The Language. Digital Press, Burlington, Mass., 1984.

[9] P. Wadler. How to replace failure by a list of successes. In Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201, pages 113--128. Springer Verlag, 1985.

[10] P. Wadler. Monads for Functional Programming In Advanced Functional Programming , Springer Verlag, LNCS 925, 1995.