A. **LE LANGAGE**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur ? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur ?

1 NOMMAGE DES IDENTIFIANTS

CONTRAINTES

En Haskell, la casse du premier caractère d'un identifiant a une importance.

Type d'identifiant	Initiale haut de casse	Initiale bas de casse
Variable		■
Fonction		■
Type	■	
Constructeur	■	

CONVENTIONS

2 TYPES DE DONNÉES

TYPES ÉLÉMENTAIRES

Haskell fournit un grand nombre de types élémentaires, dont les plus importants sont résumés dans ce tableau :

Type	Description	Bits	Bounded	Floating	Fractional	Integral	Num	Real
`Double`	Virgule flottante, double précision			■	■	■	■	■
`Float`	Virgule flottante, simple précision			■	■	■	■	■
`Int`	Entier signé à précision fixe, intervalle minimum $[-2^{29}; 2^{29}-1]$	■	■			■	■	
`Int8`	Entier signé de 8 bits	■	■			■	■	
`Int16`	Entier signé de 16 bits	■	■			■	■	
`Int32`	Entier signé de 32 bits	■	■			■	■	
`Int64`	Entier signé de 64 bits	■	■			■	■	
`Integer`	Arbitrary precision signed integer	■				■	■	
`Rational` ou `Ratio a`	Nombre rationnel de précision arbitraire				■	■	■	■
`Word8`	Entier non signé de 8 bits	■	■			■	■	
`Word16`	Entier non signé de 16 bits	■	■			■	■	
`Word32`	Entier non signé de 32 bits	■	■			■	■	
`Word64`	Entier non signé de 64 bits	■	■			■	■	

Types numériques essentiels, d'après O'Sullivan et al. 2008, p. 145, 147

TYPES COMPOSITES

Haskell connaît deux types composites : les **tableaux** et les **n-uplets**. Le type tableau se note `[a]`, un n-uplet se note `(a, b)`, `(a, b, c)`, *etc.*. Un n-uplet a au moins deux éléments, à l'exception de `()`, le n-uplet vide qui indique qu'une fonction ne renvoie pas de valeur.

Les tableaux sont typés : un tableau ne peut contenir des éléments que d'un type unique, pleinement paramétrisé. Par exemple, un tableau de type `[(Integer, Char)]` (tableau de couples `Integer`, `Char`) ne peut pas contenir un autre type de n-uplet, par exemple `(Integer, String)`.

3 DÉFINITION DE TYPES

La définition de nouveaux types se fait avec le mot-clé `data`.

Un type peut combiner plusieurs autres types sous une forme proche de celle d'un enregistrement ou d'un tuple[??] (comme un `struct` en C) et/ou fournir une alternative entre plusieurs « sous-types » (comme une `union`) qui ont chacun leur propre constructeur.

Les types de Haskell sont algébriques, c'est-à-dire qu'ils sont définis en enveloppant d'autres types dans des constructeurs, sous la forme d'une somme et/ou d'un produit. La somme de plusieurs types est une *alternative* entre ces types, leur produit est leur combinaison dans un enregistrement.

COMBINAISON DE CHAMPS

Pour décrire un livre, on peut imaginer un type qui réunirait une chaîne de caractères (le titre), un tableau de chaînes de caractères (les noms des auteurs) et un nombre entier (l'année de publication). Un tel type se déclare comme suit :

phrase intro

```
data Book = NewBook String [String] Int
```

Cette ligne définit un type nommé `Book` qui fournit un unique constructeur NewBook. NewBook se comporte comme une fonction qui prend trois paramètres et renvoie un `Book` : `NewBook :: String -> [String] -> Int -> Book`.

Pour construire un nouveau `Book`, on écrit donc `book = NewBook "Critique of Pure Reason" ["Immanuel Kant"] 1781`.

Dans cette syntaxe, les arguments du constructeur sont positionnels et doivent être fournis dans l'ordre de la déclaration.

Il est légal et très courant, quand un type ne fournit qu'un seul constructeur, de donner à ce constructeur le nom du type. On aurait alors `data Book = Book String [String] Int`. L'exemple précédent les distingue par souci de clarté, mais n'est pas vraiment idiomatique.

ALTERNATIVE ENTRE CONSTRUCTEURS

Un type algébrique présente une alternative en offrant plusieurs constructeurs.

```
data Bool = True | False
data Maybe a = Nothing | Just a
```

SYNTAXE D'ENREGISTREMENT

record syntax

La syntaxe d'enregistrement permet de nommer les champs.

```
data Book = Book {
    bookTitle :: String      -- bookTitle :: Book -> String
    bookAuthors :: [String], -- bookAuthors :: Book -> [String]
    bookYear :: Int,         -- bookISBN :: Book -> Int
    }
```

Un type qui utilise cette syntaxe peut être instantié avec des arguments positionnels ou des arguments nommés. Ces derniers peuvent être fournis dans n'importe quel ordre :

```
crp = Book "Critique de la Raison Pure" ["Immanuel Kant"] 1781
tlp = Book {
            bookYear=1921,
            bookAuthors=["Ludwig Wittgenstein"],
            bookTitle="Tractatus Logico-Philosophicus"
          }
```

Il définit automatiquement une fonction accesseur pour chacun de ses champs. Le type Book ci-dessus fournit ainsi trois fonctions `bookYear :: Book -> Int`, `bookAuthors :: Book -> [String]` et `bookTitle :: Book -> String` :

```
ghci> bookYear tlp
1921
```

Enfin, il permet de construire une nouvelle valeur à partir des champs d'une valeur existante :

```
rp = tlp {bookTitle = "Recherches philosophiques", bookYear=1953}
```

CAS PARTICULIERS

TYPES ÉNUMÉRÉS

TYPES RÉCURSIFS

Un type peut faire référence à lui-même. On peut construire un type liste identique au type natif de la façon suivante :

```
data List a = Empty | Cons a (List a)
list = (Cons 1 (Cons 2 (Cons 3 Empty)))
```

Un arbre binaire :

```
data BTree a = Node a (BTree a) (BTree a) | Empty deriving Show
```

4 SYNONYMES DE TYPES

Haskell permet de définir des synonymes pour des types existants. Les synonymes de type permettent d'augmenter la lisibilité du code ou de masquer des détails d'implémentation.

Contrairement aux types définis avec `data`[3], les informations des synonymes ne sont pas conservées à la compilation.

SYNONYMES SIMPLES AVEC TYPE

`type` crée un synonyme d'un type existant. Le synonyme et le type auquel ils renvoient sont interchangeables.

```
type ObjectId = Int16
```

Les synonymes créés avec **`type`** peuvent servir :

- À clarifier le sens des champs dans les types personnalisés sans accesseurs (**`type ISBN = Int`** pour un type `Book`, par exemple).
- Comme notation abrégée pour des types complexes fréquemment utilisés.

```
type Authors = [String]
type Title = String
type ISBN = Int
type Year = Int
data Book2 = Authors Title Year ISBN
```

DUPLICATION AVEC NEWTYPE

Le mot-clé `newtype` permet de dupliquer un type, et crée un type absolument distinct de l'original. Les synonymes créés avec `newtype` ne sont pas substituables avec le type dont ils sont synonymes. De plus, il n'appartiennent pas automatiquement aux types de classe[5] de ce dernier.

Leur syntaxe est très proche de celle de `data` :

```
newtype MyType = MyType Int
```

■ **Contrairement à `type`,** `newtype` ne maintient pas la substituabilité du nouveau type et du type dont il est un synonyme. `type` sert à faciliter la lecture, `newtype` est plutôt utilisé pour masquer l'implémentation.

- **Contrairement à `data`,** `newtype` a) n'autorise qu'un seul constructeur, b) ne conserve pas les informations du type après la compilation. Dans le programme compilé, `MyType` ci-dessus est traité comme un simple `Int` :

- **Usages de `newtype`.** Par exemple : une librairie multiplateforme fournit un accès à une ressource quelconque. Sous un des systèmes cibles, cette ressource est identifiée par un `Int64`, sous un autre, par un `String`. En utilisant **`newtype`**, on peut masquer le type sous-jacent. On pourrait parvenir au même résultat avec un type algébrique dont on exporterait pas le constructeur, mais serait plus coûteux en terme de performance et d'usage mémoire.

qsee

≠ data/newtype - question de la performance RWH 157 + conséquences sur les motifs RWH 158

5 CLASSES DE TYPE (*TYPE CLASSES*)

Les classes de type ne sont pas des classes au sens que ce terme possède en POO. Elles sont plus proches de ce qu'on nomme des interfaces : elles décrivent des fonctions pour lesquelles un type qui appartient à la classe fournit une implémentation.

DÉRIVATION AUTOMATIQUE

Les types crées avec `data` et `newtype` peuvent dériver automatiquement certains classes de type[5].

- **Avec `data`**

6 DÉRIVATION MANUELLE

```haskell
class EvalToBool a where
    toBool :: a -> Bool

instance EvalToBool Integer where
    toBool x = x /= 0
```

7 STRUCTURES CONDITIONNELLES

Haskell connaît deux structures conditionnelles : les tests binaires avec `if`, et les cas de `case`.

IF ... THEN ... ELSE

Une clause `if` est une expression, pas une structure de contrôle. La syntaxe est `if a then b else c`, où `a` est une expression de type `Bool`, `b` et `c` des expressions d'un type quelconque. Si `a` est vraie, l'expression vaut `b`, sinon `c`.

Comme c'est une expression, on peut affecter son résultat directement à une variable :

```
a = if even x then "pair" else "impair"
```

On peut souvent éviter d'employer if en utilisant des gardes[14.4], plus lisibles :

```
compte :: String -> String -> Int -> String
compte sing plur nb = if nb == 0 then "Aucun " ++ sing else if
 ↪  nb == 1 then "Un "++sing else show nb ++ " " ++ plur
```

CASE

8 ÉVALUATION PARESSEUSE

```haskell
let a = [1..] -- a est la liste de l'ensemble des entiers positifs
let b = map ((^^) 2) a
```

L'évaluation paresseuse a un prix, qui est une plus grande consommation de mémoire : au lieu d'évaluer `2 + 2`, Haskell stocke un **thunk***, c'est à dire en gros un calcul différé. Mais sur les gros traitements récursifs, l'accumulation de thunks peut entrainer rapidement un débordement de mémoire. La commande seq force l'évaluation et permet d'éviter un débordement de mémoire.

L'évaluation paresseuse obéit à des règles strictes.

Il est possible de déterminer avec précision *si* une expression va être évaluée, et si oui *quand*. C'est parce qu'il est garanti qu'une expression dont le résultat n'est pas utilisé ne sera pas évaluée qu'on peut, par exemple, programmer des opérateurs logiques court-circuitants directement en Haskell, ou manipuler des suites infinies.

9 POLYMORPHISME

POLYMORPHISME PARAMÉTRIQUE

N'importe quelle fonction[12] ou type[3] peut accepter des paramètres d'un type non défini. Sa signature remplace dans ce cas le nom d'un type par un paramètre de type, qui commence par une minuscule[1].

■ **Types polymorphiques** Le type `Maybe`, qui représente une valeur possible, est un exemple de type polymorphique. Il a deux constructeurs : `Nothing` et `Just a`. `Nothing` ne prend pas de paramètre, et représente l'absence de valeur. `Just a` prend un paramètre du type quelconque `a`.

```
ghci>:type Just 3
Just 3 :: Num a => Maybe a
ghci>:type Just "Une chaîne"
Just "Une chaîne" :: Maybe [Char]
ghci>:type Nothing
Nothing :: Maybe a
```

■ **Fonctions polymorphiques** Une fonction peut accepter, ou renvoyer, des types non-définis.

```
third :: [a] -> Maybe a
third (_:_:x:_) = Just x
third _ = Nothing
```

« Théorèmes gratuits »

Comme une fonction polymorphique n'a pas accès au type réel de son paramètre, on peut déduire (au sens strict) ce qu'elle peut faire à sa seule signature.
La fonction `head :: [a] -> a` n'a pas accès au type `a`, et par conséquent ne peut ni construire un nouvel `a`, ni modifier un des `a` du tableau qu'elle reçoit : elle doit en renvoyer un tel quel. On peut donc déduire que `head b` `` `elem` b``.
La fonction `fst :: (a, b) -> a` ne peut *rien* faire d'autre que renvoyer le premier élément de la paire qui lui est passée, et ignorer le second.

Wadler (1989) explicite le soubassement logico-mathématique de ce principe et montre des applications à des cas beaucoup plus complexes que ces quelque exemples.

10 MODULES

Haskell dispose d'un mécanisme d'importation de modules.

ÉCRIRE UN MODULE

Un module a le même nom que le fichier .hs qui le contient, et ce nom commence par une majuscule[1]. La déclaration de module a la syntaxe suivante :

```haskell
-- MyModule.hs
module Mod
    (
      x,
      y,
      z
    ) where
-- code
```

Cette déclaration exporte les identifiants x, y et z du code qui la suit. On exporterait la totalité des noms en enlevant la parenthèse, et aucun en la laissant vide.

Exporter un type sans constructeurs + lien depuis defining-types. Handle est un bon exemple de pourquoi. Aussi RWH 159

IMPORTATION DE MODULES

```haskell
-- Commande                          Importé
import Mod                        -- x, y, z, Mod.x, Mod.y, Mod.z
import Mod ()                     -- (Nothing!)
import Mod (x,y)                  -- x, y, Mod.x, Mod.y
import qualified Mod              -- Mod.x, Mod.y, Mod.z
import qualified Mod (x,y)        -- Mod.x, Mod.y
import Mod hiding (x,y)           -- z, Mod.z
import qualified Mod hiding (x,y) -- Mod.z
import Mod as Foo                 -- x, y, z, Foo.x, Foo.y, Foo.z
import Mod as Foo (x,y)           -- x, y, Foo.x, Foo.y
import qualified Mod as Foo       -- Foo.x, Foo.y, Foo.z
import qualified Mod as Foo (x,y) -- Foo.x, Foo.y
```

D'après Hudak et al. 2000

11 LE PRÉLUDE

Le Prélude (`Prelude`) est la librairie fondamentale d'Haskell. Contrairement aux autres modules, il est importé implicitement (cette importation peut néanmoins être contrôlée avec une clause `import`[10.2] explicite).

L'implémentation de référence (Marlow 2010, I, 9) est écrite en Haskell.

Il est particulièrement intéressant de noter que parmi les définitions fournies par le Prélude, un certain nombre sont, dans la plupart des langages procéduraux, définies au niveau du compilateur. Parmi celles-ci, on trouve notamment les opérateurs booléens court-circuitants, dont l'implémentation est rendue triviale par le principe d'évaluation paresseuse :

- Les opérateurs booléens court-circuitants :

```haskell
module Prelude (
    module PreludeList, module PreludeText, module PreludeIO,
    Bool(False, True),
    Maybe(Nothing, Just),
    Either(Left, Right),
    Ordering(LT, EQ, GT),
    Char, String, Int, Integer, Float, Double, Rational, IO,

--      These built-in types are defined in the Prelude, but
--      are denoted by built-in syntax, and cannot legally
--      appear in an export list.
--  List type: []((:), [])
--  Tuple types: (,)((,)), (,,)((,,)), etc.
--  Trivial type: ()(())
--  Functions: (->)

    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Fractional((/), recip, fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase, sin, cos, tan,
             asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh),
    RealFrac(properFraction, truncate, round, ceiling, floor),
    RealFloat(floatRadix, floatDigits, floatRange, decodeFloat,
              encodeFloat, exponent, significand, scaleFloat, isNaN,
              isInfinite, isDenormalized, isIEEE, isNegativeZero, atan2),
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    maybe, either,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, realToFrac,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined,
    seq, ($!)
  ) where
```

Listing 1 : Noms exportés par le Prélude d'Haskell 2010 (Marlow 2010, I, 9)

B. **FONCTIONS**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur ? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur ?

12 FONCTIONS ET VARIABLES

Haskell n'a pas de notion de variable au sens qu'a ce terme en programmation procédurale. Il est possible d'assigner une expression ou une valeur à un nom, avec la syntaxe `nom = expression`, mais `nom` est immuable, et est donc plus proche d'une constante (c'est une variable au sens mathématique du terme).

En combinant ceci avec les principes de **transparence référentielle***, d'évaluation paresseuse[8] et d'application partielle[??], on voit facilement qu'il n'existe aucune différence stricte entre une fonction et une variable, donc qu'il n'existe pas de variables. Par exemple :

```
a = 3 * 2
times3 x = 3 * x
b = times3 2
c = 6
```

Ici, `times3` est une fonction, `a`, `b` et `c` des variables. Dans la mesure où la valeur d'aucune n'est évaluée tant qu'elle n'est pas utilisée, la variable `a` a strictement la même valeur que `b`, qui n'est pas 6, mais le **thunk*** `3 * 2`.

> ⚠ Cette identité n'est vraie que des fonctions pures. Les fonctions impures, comme par exemple `getLine`, peuvent évidemment renvoyer un résultat différent à chaque invocation. Voir **??** : **??**.

La suite de cette fiche ne s'intéresse donc qu'aux fonctions, puisque les « variables » n'en sont qu'un cas particulier.

SIGNATURE DE TYPE

La signature a la forme `f :: TypeA -> TypeRet`, ce qui signifie que la fonction prend un paramètre de type `TypeA` et renvoie une valeur de type `TypeRet`.

Une fonction définie avec plusieurs paramètres a pour signature `f :: TypeA -> TypeB -> TypeC -> TypeRet`. Cette syntaxe est explicitée fiche 15 : **Application partielle et *currying*** .

Les fonctions d'ordre supérieur utilisent les parenthèses pour indiquer qu'elles prennent une autre fonction en paramètre. Par exemple, le type `map :: (a -> b) -> [a] -> [b]` se lit : `map` prend comme premier paramètre une fonction quelconque `x :: a -> b`.

Une variable ou une fonction sans paramètres a pour type `nom :: Type`.

FONCTIONS PRÉFIXES ET INFIXES

Une fonction est dite préfixe si son nom est placé avant ses arguments, et infixe si son nom est placé entre ses arguments. `map` est une fonction préfixe, `+` est infixe. La

distinction est syntaxique, et se fait au niveau des caractères qui constituent le nom de la fonction.

- **Une fonction infixe** a un nom composé uniquement de symboles non alphanumériques : `+`, `*` ou `>>=` sont infixes.

  On peut utiliser une fonction infixe comme préfixe en entourant son nom de parenthèses : `(+) 1 1`.

- **Une fonction préfixe** a un nom composé de caractères alphanumériques. `map`, `elem` ou `foldr` sont préfixes.

  On peut utiliser une fonction préfixe comme infixe en entourant son nom de *backticks* : `1 `elem` [1..10]`.

13 DÉFINITION DE FONCTIONS

Une fonction se définit de la façon suivante :

```
add :: a -> b  -- Signature de type, généralement optionnel.
add x = expr x
```

Une fonction infixe se définit en entourant son nom de parenthèses, comme pour l'utiliser en préfixe :

```
(+*) a b = a + b + a * b
```

FONCTIONS LOCALES

On peut définir des fonctions dont la visibilité est limitée à une fonction. C'est utile pour définir des constantes, ou fournir des fonctions utilitaires qui n'ont pas besoin d'être disponibles au niveau du module. Haskell propose deux syntaxes : **`let`**, qui place les variables locales *avant* le code de la fonction, et **`where`**, qui les positionne *après*.

```
circLet :: Fractional a => a -> a
circLet radius = let pi   = 3.14
                     diam = 2 *
                      ↪  radius
                 in pi * diam
```

```
circWhere :: Fractional a => a -> a
circWhere radius = pi * diam
    where pi   = 3.141592653589793
          diam = 2 * radius
```

- Le choix de l'une ou de l'autre syntaxe est une question de lisibilité.
- On peut les imbriquer : une fonction définie dans une clause **`let`**/**`where`** peut à son tour définir des fonctions locales, etc.
- La visiblité des fonctions locales est limitée à la définition englobante.

FIXITÉ (PRÉCÉDENCE ET ASSOCIATIVITÉ)

L'associativité et la précédence sont collectivement nommées « fixité ». La fixité d'une fonction infixe (et de n'importe quelle fonction préfixe dans sa forme infixe, comme `` `elem` ``) est fixée par une déclaration **`infixl`** (associatif à gauche), **`infixr`** (associatif à droite) ou **`infix`** (non-associatif), suivie de l'ordre de précédence compris entre 0 et 9 et du nom de la fonction :

```
(+*) :: Num a => a -> a -> a
infixl 9 +*
(+*) a b = a + b + a * b
```

Il est possible de définir la fixité d'une fonction locale, directement dans la clause **`let`** ou **`where`** où elle est définie.

14 PARAMÈTRES, MOTIFS ET GARDES

PASSAGE DE PARAMÈTRES

- **Déconstruction de types composites.** « Déconstruire » un argument d'une fonction permet d'obtenir directement les arguments du constructeur. Par exemple, la fonction suivante déconstruit un constructeur de paire (tuple de deux éléments) pour en renvoyer le premier :

```
toggle :: (a, b) -> a
toggle (x, y) = (y, x)
```

Un paramètre non utilisé peut être remplacé par un `_` :

```
duplFirst :: (a, b) -> (a, a)
duplFirst (x, _) = (x, x)
```

On n'a pas besoin du second membre de la paire : on la décompose donc en évitant de nommer cet élément.

De la même façon, si le paramètre est un `Maybe`, on peut récupérer directement sa valeur en déconstruisant `Just` :

```
double :: Maybe Int -> Int
double (Just x) = x * 2
```

- **Motifs nommés.** On peut avoir besoin de déconstruire un paramètre selon un motif en conservant le paramètre entier. Les motifs nommés permettent d'éviter des suites déconstruction-reconstruction redondantes.

La fonction `suffixes` (O'Sullivan et al. 2008, p. 103) renvoie tous les suffixes d'une liste. Elle peut s'écrire :

```
suffixes :: [a] -> [[a]]
suffixes xs@(_:xs') = xs : suffixes xs'
suffixes _ = []
```

FILTRAGE PAR MOTIF ET GARDES

Le filtrage par motifs et l'emploi de gardes permettent de proposer différentes implémentations d'une même fonction selon les paramètres qui y sont passés, de façon similaire à l'emploi de cas en notation mathématique :

$$f(x) = \begin{cases} f(x-1) + x & \text{si } x > 0 \\ 1 & \text{sinon} \end{cases}$$

Le filtrage par motifs permet de choisir une implémentation selon le type et dans une certaine mesure la valeur des paramètres, les gardes selon une expression arbitraire.

Le filtrage par motif et les gardes permettent de définir plusieurs cas qui se recouvrent. Par exemple, une fonction peut fournir une implémentation pour n'importe quelle liste, et une autre pour n'importe quelle liste *non vide*. Haskell utilise toujours la première implémentation qui s'applique aux paramètres, dans l'ordre de déclaration : il faut donc déclarer les moins générales en premier.

FILTRAGE PAR MOTIFS

Le filtrage par motifs permet de filtrer selon un constructeur ou selon une valeur arbitraire.

- **Par constructeur.** Le filtrage par constructeurs permet de sélectionner quel constructeur d'un type algébrique[3.2] correspond à quelle implémentation.

```haskell
maybeIntToStr :: Maybe Int -> String
maybeIntToStr (Just a) = show a
maybeIntToStr Nothing  = "NaN"
```

```haskell
mySum :: (Num a) => [a] -> a
mySum (x:xs) = x + mySum xs
mySum []     = 0
```

- **Par valeur littérale.** Le filtrage par valeur littérale est le plus simple. Il choisit une implémentation si un paramètre a une valeur déterminée.

```haskell
compte :: String -> String -> Int -> String
compte singulier pluriel 0 = "Aucun(e) " ++ singulier
compte singulier pluriel 1 = "Un(e) " ++ singulier
compte singulier pluriel quantite = show quantite ++ " " ++  pluriel
```

Une valeur littérale *doit* être littérale et ne peut pas, pour des raisons syntaxiques, être une variable. Ce code est erroné :

```
1  ultimate =
    ↪  "the Ultimate Question of Life, the Universe, and Everything"
2
3  answer :: String -> Int
4  answer ultimate = 42
5  answer _ = 0
```

`ultimate` ligne 4 n'est *pas* la variable qui a été définie ligne 1, mais une variable locale qui capture n'importe quel paramètre passé à la fonction et qui masque la variable globale `ultimate`. Les lignes 4–5 ne fonctionneront pas comme on pourrait s'y attendre : elles sont strictement équivalentes à la ligne unique `answer _ = 42`.
Pour employer des constantes nommées, on peut utiliser un préprocesseur [??] ou les remplacer par un type algébrique[3.2].

■ **Paramètres ignorés.** Certaines implémentations d'une fonction peuvent ne pas faire usage de tous les paramètres. On ignore un paramètre dans la définition avec le symbole `_` :

La fonction `compte` ci-dessus pourrait s'écrire :

```
compte :: String -> String -> Int -> String
compte singulier _ 0 = "Aucun(e) " ++ singulier
compte singulier _ 1 = "Un(e) " ++ singulier
compte _ pluriel quantite = show quantite ++ " " ++  pluriel
```

`_` n'est pas un nom de variable mais la mention explicite que le paramètre ne sera pas utilisé.

GARDES

Un garde est une expression de type `Bool`. Si l'expression s'évalue à `True`, l'implémentation qui suit est utilisée.

Leur syntaxe est :

```
func args | garde = impl
```

Par exemple, une fonction qui détermine si un nombre est pair, qui s'implémenterait naïvement sous la forme `isEven x = if x `mod` 2 == 0 then True else False` peut s'écrire plus lisiblement :

```
isEven x | x `mod` 2 == 0 = True
isEven _ = False
```

La partie à gauche du garde peut être omise si elle est identique à celle qui précède (c'est-à-dire si l'éventuel motif est le même) :

```haskell
isEven x | x `mod` 2 == 0 = True
         | otherwise = False
```

`otherwise` est une constante définie dans le Prélude. Sa valeur est simplement `True`.

⚠ `otherwise` est simplement définie comme `otherwise = True`. Son emploi est donc limité aux gardes.

« *PATTERN GUARDS* »

Haskell 2010 or `PatternGuards` extension.

Haskell 2010 étend la syntaxe des gardes

Cette section

```haskell
gardes :: Int -> String
gardes a | odd a, a `mod` 5 == 0 = "Impair et/ou multiple de 5"
         | even a = "Pair mais pas multiple de 5"
```

15 APPLICATION PARTIELLE ET *CURRYING*

Une fonction, quel que soit le nombre de paramètres avec lequel elle a été déclarée, ne prend qu'un seul paramètre et renvoie une autre fonction. Le type de `+`, par exemple, est : `Num a => Num a -> Num a -> Num a`, ce qui signifie que `+` prend un premier paramètre d'un type de type `Num`

16 LAMBDAS

C. MANIPULATION DE DONNÉES

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur ? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur ?

17 OPÉRATIONS SUR DES LISTES

D. ENTRÉES/SORTIES

La gestion des entrées/sorties requiert un traitement spécifique dans un langage fonctionnel. Contrairement aux fonctions pures du langage, les fonctions d'E/S produisent des effets de bord, et violent le principe de **transparence référentielle***.
Le mécanisme d'E/S d'Haskell est implémenté sous la forme d'une monade[23] nommée `IO`.

18 FONCTIONS D'ENTRÉE/SORTIE

FONCTIONS D'ENTRÉE

Prelude	h*	Fonction		Description
■	■	`getChar`	`:: IO Char`	Lit un caractère.
■	■	`getLine`	`:: IO String`	Lit une ligne.
■	■	`getContents`	`:: IO String`	Lit le contenu d'un fichier.

FONCTIONS DE SORTIE

MANIPULATION DE FICHIERS OU DE RÉPERTOIRES

19 GESTION DES ERREURS

LES TYPES `MAYBE` ET `EITHER`

EXCEPTIONS

E. **IDIOMES**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur ? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur ?

21 **RÉCURSIVITÉ**

RÉCURSIVITÉ EN QUEUE

FOLDS

22 MONOÏDES

23 MONADES

F. **OUTILS**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur ? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur ?

25 DEBUGGERS

27 NOTIONS À INTÉGRER

À intégrer, en vrac :

- Idiome : Point-free style (RWH 120)
- Lexique : Liste de paires = association list (RWH 121)
- Extensions :
  - TypeSynonymInstances
  - OverlappingInstances
- monomorphisme (RWH 163, Haskell 98 4.5.5)
- IO
  - Qu'est ce qu'une action (RWH 167, 184)
  - Buffering (RWH 189)
  - Data.ByteString, Data.ByteString.Lazy

Références

Hudak, Paul, John Peterson et Joseph Fasel (2000). *A Gentle Introduction to Haskell. Version 98*.
url : https://www.haskell.org/tutorial/index.html.

Marlow, Simon, éd. (2010). *Haskell 2010 Language Report*.
url : https://wiki.haskell.org/Language_and_library_specification#The_Haskell_2010_report.

O'Sullivan, Bryan, Don Stewart et John Goerzen (2008). *Real World Haskell*. O'Reilly.
url : http://book.realworldhaskell.org/.

Wadler, Philip (1989). *Theorems for Free !*
url : http://www.cs.sfu.ca/CourseCentral/831/burton/Notes/July14/free.pdf.

Glossaire

thunk

Un thunk est une expression non encore évaluée. .

expliciter sémantique non stricte, faire lien avec éval paresseuse

transparence référentielle

Une propriété des expressions pures telle que toute expression peut être remplacée par son résultat sans modifier le comportement du programme. À l'inverse, les expressions impures sont dites sont référentiellement opaques..

Table des matières