Ecole d'Ingénieurs du Canton de Vaud

Programmation fonctionnelle en Haskell

Claude Evéquoz

Mai 2005

Pourquoi faire de la programmation fonctionnelle?

- Illustrer le paradigme et notamment comprendre comment une programmation sans variable et affectation peut se faire élégamment (programmation sans effets de bord);
- Comprendre les mécanismes avancés de manipulation de fonctions;
- Voir qu'il est possible de faire une analyse fonctionnelle d'un cahier des charges compilable et exécutable, c.-à-d. programmer dans un langage de spécification;
- Mieux appréhender l'évolution des langages impératifs car beaucoup de notions fonctionnelles sont reprises par ces langages.

Propriétés des langages fonctionnels

Pas d'effets de bord : Puisqu'il n'y a pas de variables, les problèmes des variables globales, des alias, des pointeurs disparaissent; et, 2 évaluations de la même fonction donnent toujours le même résultat.

Traitement des fonctions comme des valeurs : Les fonctions peuvent être passées en paramètre et peuvent retourner d'autres fonctions.

Transparence référentielle : Tout identificateur peut être substitué par sa valeur (et vis versa) car les expressions peuvent s'évaluer à n'importe quel instant; les raisonnements de manipulation algébrique s'appliquent.

Propriétés additionnelles de Haskell

- *Fortement typé*
- *Inférence de typage* : Le compilateur détermine lui-même les types des fonctions.
- *Polymorphisme algorithmique* : Un algorithme peut s'écrire sans tenir compte des propriétés spécifiques des données qui sont transformées.
- *Instantiation partielle de fonction* : Former de nouvelles fonctions à partir d'autres en fixant certains paramètres.
- *Évaluation paresseuse des arguments* : Introduit la possibilité de réaliser des structures de données infinies.
- *Ensembles ZF* (Zermelo-Fränkel) : Listes infinies ayant certaines propriétés.
- Traitement uniforme d'actions purement impératives par la notion de *monade*.
- *Classes de types* : Regroupement des types en classes ayant les mêmes propriétés. Par exemple on peut additionner 2 entiers, 2 réels, mais pas 2 caractères. Si une classe numérique est définie, les entiers et les réels devraient appartenir à cette classe.
- Possibilité d'invoquer des fonctions écrites dans un autre langage.

Où sont les problèmes?

Difficulté de compréhension : La programmation fonctionnelle est certes plus abstraite que le carcan de la programmation impérative, et donc certainement plus difficile.

Perte de performance : Le code généré par les compilateurs Haskell est quelque peu plus lent comparé à une implémentation entièrement en C. Mais d'autres langages fonctionnels, par exemple ML (semblable à Haskell), produisent des exécutables souvent plus rapides que leur pendant C++.

Motivation

Les 3 fonctions ci-après définissent respectivement $n!$, $\sum_{i=0}^{n} i$ et $\sum_{i=0}^{n} i^2$

```
fact n | n == 0 = 1                -- 0! = 1
       | n > 0 = n * fact (n-1)    -- n! = n * (n-1)!

sommeEntier n | n == 0 = 0
              | n > 0 = n + sommeEntier (n-1)

sommeCarre n | n == 0 = 0
             | n > 0 = n * n + sommeCarre (n-1)
```

Ces 3 fonctions utilisent le même principe de récurrence : une condition d'arrêt ($n == 0$) et un pas inductif exprimant la relation entre n et $n-1$. Ce principe peut se définir par une autre fonction plus générale :

```
recursion arret induction n
 | n == 0 = arret
 | n > 0 = induction n (recursion arret induction (n-1))
```

La 1ère équation donne la condition d'arrêt, et la seconde indique comment il faut combiner n avec le résultat précédent, c.-à-d. appliquer la fonction `induction`.

À présent $n!$, $\sum_{i=0}^{n} i$ et $\sum_{i=0}^{n} i^2$ peuvent s'écrire

```
fact n = recursion 1 (*) n

sommeEntier n = recursion 0 (+) n
```

`(*)` et `(+)` sont les fonctions préfixées des opérateurs `*` et `+`.

Motivation

$\sum_{i=0}^{n} i^2$ est à peine plus difficile, mais il faut avant définir une fonction qui accumule des valeurs élevées au carré :

```
f x y = x * x + y
sommeCarre n = recursion 0 f n
```

Vérifions que notre dernière définition de `sommeCarre` correspond à ce que nous recherchons.

Pour $n = 0$,

```
sommeCarre 0 = recursion 0 f 0
≡ 0
```

Pour $n > 0$,

```
sommeCarre n = recursion 0 f n
≡ f n (recursion 0 f (n-1))
≡ f n sommeCarre (n-1)
≡ n * n + sommeCarre (n-1)
```

Ce qui correspond à notre 1ère définition de `sommeCarre`.

Remarques

(1) La manipulation se fait en utilisant la définition des fonctions sans les transformer.

(2) Les substitutions sont toujours valides car les fonctions n'ont pas d'effet de bord.

(3) Le résultat d'une fonction dépend uniquement de ses paramètres et l'ordre des évaluations des arguments n'influence pas le résultat final.

Environnements de programmation Haskell

Il existe 3 principaux environnements Haskell :

HUGS : environnement interactif extrêmement populaire;

GHC : environnement interactif (*GHCi*) plus lent que *HUGS* mais ayant un compilateur (*GHC*);

NHC : compilateur produisant des exécutables plus petit et souvent plus rapide que *GHC*.

Exemple d'une session interactive Haskell (GHCi)

(1) Lancement de la session

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.2, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude>
```

(2) Définition et évaluation de la fonction `sommeCarre`

```
Prelude> let sommeCarre n | n == 0 = 0 | n > 0 = n * n + sommeCarre (n-1)
```

(3) Application de `sommeCarre`

```
Prelude> sommeCarre 2
5
```

(4) Détermination du type de `sommeCarre`

```
Prelude> :t sommeCarre
sommeCarre :: forall a. (Num a, Ord a) => a -> a
```

(5) Fin de la session

```
Prelude> :q
Leaving GHCi
```

Environnements de programmation Haskell

Exemple d'une session interactive Haskell (GHCi) avec chargement d'un programme

(1) Création d'un fichier

```
C:\WINDOWS>d:
D:\My Documents>cd Haskell
D:\My Documents\Haskell>type exemple.hs
recursion arret induction n
    | n == 0 = arret
    | n > 0 = induction n (recursion arret induction (n-1))
f x y = x * x + y
sommeCarre n = recursion 0 f n
```

(2) Lancement de la session

```
D:\My Documents\Haskell>ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.2, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude> :cd d:/mydocu~1/haskell
Prelude> :l exemple
Compiling Main             (exemple.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

(3) Utilisation des fonctions chargées

```
*Main> f 2 3
7
*Main> sommeCarre 2
5
*Main>
```

Constantes en Haskell

Constantes entières

En Haskell, il y a 2 types différents pour exprimer un nombre entier : les entiers à taille fixe (*Int*) et les entiers à précision variable (*Integer*).

Ces constantes peuvent être introduites en base 8, 10 ou 16.

Syntaxe :

<*entier*> ::= (<*décimal*> | `0`(o|O)<*octal*> | `0`(x|X) <*hexadécimal*>)

<*décimal*> ::= <*chiffre10*> {<*chiffre10*>}*

<*octal*> ::= <*chiffre8*> {<*chiffre8*>}*

<*hexadécimal*> ::= <*chiffre16*> {<*chiffre16*>}*

<*chiffre8*> ::= `0` | `1` | ... | `7`

<*chiffre10*> ::= <*chifffre8*> | `8` | `9`

<*chiffre16*> ::= <*chifffre10*> | a | `A` | b | `B` | c | `C` | d | `D` | e | `E` | `f` | `F`

Selon la norme Haskell98, un *Int* doit être au moins dans l'intervalle $[-2^{29},2^{29}-1]$. Sur des processeurs 32, l'intervalle est évidemment équivalent aux entiers signés.

Constantes en Haskell

Constantes réelles

Comme en C, il y des réels définis sur 32 bits (*Float*) et sur 64 bits (*Double*).

Selon la norme Haskell98, l'intervalle des réels doit suivre l'intervalle recommandé par l'IEEE, toutefois les débordements (NaN, +Inf, etc) n'ont pas l'obligation d'être implémentés.

Syntaxe :

<*réel*> ::= <*décimal*> `.` <*décimal*> [<*exposant*>] |
<*décimal*> <*exposant*>

<*exposant*> ::= (e|E) [+|-]<*décimal*>

Constantes booléennes

Il y a deux valeurs prédéfinies pour les booléens (*Bool*) :

Syntaxe :

<*booléen*> ::= `True` | `False`

Contrairement à d'autres langages, Haskell fait une distinction entre les minuscules et les majuscules. Ainsi `true` ou `TRUE` ne sont pas définies.

Constantes en Haskell

Constantes caractères

Les constantes caractères et la définition des chaînes de caractères sont toujours complexes à donner formellement si nous souhaitons traiter tous les cas. Haskell n'est pas une exception. Dans ce qui suit nous nous limitons à des exemples qui couvrent la majorité des cas utiles du cours.

Un caractère s'écrit entre apostrophe alors qu'une chaîne de caractères s'écrit entre guillemets :

Exemples : `'c'` et `"coucou"`

Une barre oblique inversée (`\`) permet d'introduire des caractères spéciaux :

- `\a` : alarme
- `\b` : recul vers la gauche
- `\f` : saut de page
- `\n` : nouvelle ligne
- `\r` : retour de chariot
- `\t` : tabulation horizontale
- `\v` : tabulation verticale
- `\\` : le caractère `\`

Il est possible de définir des caractères par leur code ascii ou unicode.

`\'` : le caractère `'`, nécessaire pour une constante caractère

`\"` : le caractère `"`, nécessaire pour une chaîne de caractères

Constantes en Haskell

Constantes caractères (suite)

Une longue chaîne de caractères peut s'écrire sur plusieurs lignes :

```
"Ceci est une \
\longue chaine"
```

Remarquons qu'il n'est généralement pas possible d'exécuter l'exemple précédent sur un interpréteur Haskell (p. ex. *GHCi*), car lors de l'introduction d'une nouvelle ligne, l'interpréteur évalue la ligne alors qu'elle n'est pas terminée.

Commentaires

Il est possible d'annoter du code Haskell par des commentaires.

- Un commentaire de ligne s'introduit par 2 caractères '`-`' consécutifs et se termine à la fin de cette ligne. (comme en Ada)
- Les commentaires peuvent aussi apparaître sur plusieurs lignes. Ceux-ci débutent par les caractères `{-` et se terminent par `-}`.

  Ces commentaires peuvent être imbriqués :

```
{- Ceci est un commentaire {-
   sur plusieurs lignes. -}-}
```

Premières difficultés à surmonter

Les paramètres d'une fonction ne nécessitent pas d'être mis entre parenthèses.

L'appel de la fonction $f(x,y)$ signifie que f est une fonction qui prend 1 paramètre qui est le produit cartésien de 2 types. Ceci est différent d'une fonction f qui prend 2 paramètres dont les types sont ceux du produit cartésien. Ainsi, on écrit $f\ x\ y$ au lieu de $f(x,y)$.

Remarquons qu'il n'y a pas de différence entre $g(x)$ et $g\ x$.

Les *opérateurs* sont des fonctions ayant des identificateurs particuliers :

<*opid*> ::= <*symbole*> {<*symbole*>|_}*

<*symbole*> ::= ! | # | $ | % | & | * | + | . | / | < | = | > | ? | @ | \ | ^ | | | - | ~

- Un opérateur peut s'utiliser comme une fonction : `(+) 3 4` et `3 + 4` sont équivalents et utilisent le même opérateur.
- Une fonction peut aussi s'utiliser comme un opérateur : `mod 10 3` et `10 `mod` 3` utilisent la même fonction `mod`.
- Les opérateurs sont classés en 10 niveaux de précédence différents, allant de 0 à 9. La précédence 9 est plus prioritaire que la précédence 0.

Expressions arithmétiques

Les opérateurs arithmétiques prédéfinis sont regroupés en 4 niveaux de précédence. Ce sont

9		négation unaire `negate`
8	droite	exponentiation `^`
7	gauche	opérateurs multiplicatifs `*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``
6	gauche	moins unaire `-` et opérateurs additifs `+ -`

- Le symbole `-` est à la fois le moins unaire et la soustraction. Exemples

```
Prelude> -2 `mod` 3
-2
Prelude> -(2 `mod` 3)
-2
Prelude> (-2) `mod` 3
1
Prelude> negate 2 `mod` 3
1
```

- `div`, `mod`, `rem` et `quot` s'appliquent uniquement à des entiers.
- `quot` est la division entière en tronquant le résultat (arrondissement vers le 0).
- `div` est la division entière en arrondissant vers -∞.
- `rem` est le reste d'une division entière.
- `/` est la division réelle.

Expressions arithmétiques

`div`, `mod`, `rem` et `quot` satisfont les relations ci-dessous :

```
(x `quot` y) * y + (x `rem` y) == x (si y ≠ 0)
(x `div` y) * y + (x `mod` y) == x  (si y ≠ 0)
```

Contrairement à C, `mod` est un véritable modulo.

```
Prelude> (-2) `mod` 3
1
Prelude> 2 `mod` (-3)
-1
Prelude> (-2) `mod` (-3)
-2
```

L'exponentiation (^) élève un nombre à une puissance entière.

```
Prelude> 3.0^2
9.0
```

L'évaluation d'une expression suit les mêmes règles que les langages Ada et C : les opérateurs les plus prioritaires sont d'abord évalués, puis, pour les opérateurs ayant la même priorité, l'évaluation se fait de la gauche vers la droite si l'opérateur est associatif à gauche, sinon dans l'autre sens pour les opérateurs associatifs à droite.

Exemples

```
Prelude> 3.0 - 3.1 - 0.1
-0.20000000000000001
Prelude> 3.0 - (3.1 - 0.1)
0.0
Prelude> 27 `div` (5 `mod` 3) * 2
26
```

Opérateurs de comparaison

Ces opérateurs sont moins prioritaires que les opérateurs arithmétiques, ils ont une précédence de 4. Ce sont les opérateurs ==, /=, <, <=, > et >=, et ils ne sont pas associatifs.

Exemple :

```
Prelude> 2 < 1 + 2
True
```

Remaques

- Ces opérateurs s'appliquent aux types *Int*, *Integer*, *Float*, *Double* et *Char*.
- Pour les chaînes de caractères, ces opérateurs réagissent de manière semblable à la fonction `strcmp` du langage C :

  `c1 < c2` est équivalent à `strcmp(c1,c2) < 0`

  `c1 > c2` est équivalent à `strcmp(c1,c2) > 0`

  `c1 == c2` est équivalent à `strcmp(c1,c2) == 0`

  `c1 <= c2` est équivalent à `strcmp(c1,c2) <= 0`

  `c1 >= c2` est équivalent à `strcmp(c1,c2) >= 0`

  `c1 /= c2` est équivalent à `strcmp(c1,c2) != 0`

Opérateurs logiques

Il y en a 3 :

9 négation booléenne `not`

3 et `&&`

2 ou `||`

- La négation booléenne est en fait une fonction, mais agit comme un opérateur (puisqu'elle n'a qu'un paramètre).
- L'expression `not 1 < 2` est dépourvue de sens et génère une erreur, par contre `not (1 < 2)` est correcte.
- Les opérateurs `&&` et `||` agissent de la même manière que les opérateurs `&&` et `||` de C ou les opérateurs `and then` et `or else` d'Ada.

Exemple

```
Prelude> 1 < 2 || 3 > 4
True
```

L'expression `3 > 4` n'est pas évaluée en Haskell.

Expressions conditionnelles

Elles ont la forme attendue :

`if` <*condition*> `then` <*partie vraie*> `else` <*partie fausse*>

Exemple

```
Prelude> if 1 < 2 then 3 + 1 else 2
4
```

Remarques

- Contrairement aux énoncés conditionnels retrouvés dans les langages procéduraux, la clause `else` n'est pas optionnelle car le résultat de l'évaluation doit toujours retourner une valeur.
- Les expressions conditionnelles en Haskell se comportent et s'utilisent de la même manière que l'opérateur ternaire ? : de C.

Exercices

Exercice 1

Donnez les résultats des expressions ci-dessous.

(1) `'Z' < 'a'`
(2) `"abc" <= "ab"`
(3) `"abc" >= "ac"`
(4) `1 + 2 * 3`
(5) `5.0 - 4.2 / 2.1`
(6) `3 > 4 || 5 < 6 && not (7 /= 8)` (dites quelles expressions sont évaluées)
(7) `if 6 < 10 then 6.0 else 10.0`
(8) `0XaB + 2`
(9) `0xBa + 2`

Exercice 2

Toutes les expressions ci-dessous comportent des erreurs. Expliquez chacun.

(1) `18 `mod` 7 / 2`
(2) `if 2 < 3 then 3`
(3) `1 < 2 and 5 > 3`
(4) `6 + 7 div 2`
(5) `4. + 3.5`
(6) `1.0 < 2.0 or 3 > 4`
(7) `1.0 = 3`
(8) `if 4 > 4.5 then 3.0 else 'a'`

Exercice 3

Écrivez les expressions ci-dessous en Haskell sans utiliser les opérateurs logiques :

(1) `E1 || E2`
(2) `E1 && E2`

Conversions explicites

Conversions explicites entre entiers et réels

La fonction `fromInteger` convertit un entier en un autre type et en particulier à un nombre réel :

```
Prelude> 7 `mod` 8 / 2
Ambiguous type variable `a' in these top-level constraints:
  `Fractional a' arising from use of `/' at <interactive>:1
  `Integral a' arising from use of `mod' at <interactive>:1
Prelude> fromInteger (7 `mod` 8) / 2
3.5
```

Il existe 4 fonctions pour convertir les réels en nombres entiers

- `floor r` donne le plus grand entier plus petit ou égal à `r`;
- `ceiling r` donne le plus petit entier plus grand ou égal à `r`;
- `truncate r` donne la partie entière du nombre `r`;
- `round r` arrondit `r`.

r	floor (r)	ceiling (r)	round (r)	truncate (r)
2.5	2	3	2	2
-2.5	-3	-2	-2	-2
2.3	2	3	2	2
-2.6	-3	-2	-3	-2

Conversions explicites

Conversion explicite entre entiers et caractères

- La fonction `ord` convertit un caractère en un nombre entier et retourne le code ASCII de son argument.

```
Prelude Char> ord 'A'
65
```

- La fonction `chr` réalise l'inverse.

```
Prelude Char> chr 65
'A'
```

Remarque

- Ces 2 fonctions sont dans un module (concept que nous verrons plus tard) appelé *Char*. Il faut alors soit qualifier le nom des fonctions par le nom du module :

  `Char.ord` ou `Char.chr`

  ou alors charger le module *Char* dans une session interactive Haskell :

```
Prelude> :module +Char
Prelude Char>
```

Exercice

(1) Convertissez `65.0` en caractère.

(2) Convertissez `'A'` en réel.

Types composés : tuples

Un *tuple* est un produit cartésien dont les champs sont anonymes.

Syntaxe :

<*tuple*> ::= (<*expression*> {,<*expression*>}$^{+}$)

Exemples

```
Prelude> (4,5.0,"six")
(4,5.0,"six")
Prelude> ('a',3,"trois")
('a',3,"trois")
```

Remarquons que les exemples ci-dessous sont différents :

- `(1,2,3)`
- `(1,(2,3))`
- `((1,2),3)`

Il existe 2 fonctions prédéfinies permettant d'extraire les éléments d'une paire :

`fst` retourne le premier élément d'une paire et `snd` le second.

```
Prelude> fst ('a',"six")
'a'
Prelude> snd ('a',"six")
"six"
```

Pour des tuples plus larges, il n'existe ni de fonction prédéfinie, ni d'opérateur. Pour extraire les éléments de ces tuples, il faut passer par un mécanisme plus général, la notion de gabarit.

Types composés : listes

Une *liste* est une collection ordonnée d'éléments ayant le même type.

Syntaxe :

<*liste*> ::= [<*expression*> {,<*expression*>}*] | []

Exemples simples :

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

- En Haskell, le type d'une liste se note par [<*type*>].

  Exemples :

  `[1,2,3]` est de type [*Integer*], c.-à-d. une liste d'entiers

  `['a','b','c']` est de type [*Char*], c.-à-d. une liste de caractères
- Une liste vide (liste comportant aucun élément) s'indique par `[]`.
- Toute liste à l'exception d'une liste vide se compose d'un élément de tête (*head*) et d'une sous-liste de queue (*tail*).
- La fonction `head` retourne l'élément de tête d'une liste.
- La fonction `tail` retourne la liste sans son élément de tête.
- Exemples

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

Types composés : listes

En Haskell, il est possible de définir des listes ayant certaines propriétés par la notation `..` (prononcée «point-point»). Les formes admissibles sont

<*liste définie*> ::= [<*expression*>] [, <*expression*>] `..` [<*expression*>]]

[a`..`b] : liste de a jusqu'à b par pas de 1

[a,b`..`c] : liste de a jusqu'à c par pas de $b–a$

[a`..`] : liste de a jusqu'à l'infinie par pas de 1

[a,b`..`] : liste de a jusqu'à l'infinie par pas de $b–a$

- Exemples

```
Prelude> [1..6]
[1,2,3,4,5,6]
Prelude> [1,2..6]
[1,2,3,4,5,6]
Prelude> [1,3..6]
[1,3,5]
Prelude> [6..0]
[]
Prelude> [6,5..0]
[6,5,4,3,2,1,0]
Prelude> ['a'..'d']
"abcd"
Prelude> [1+1..8-2]
[2,3,4,5,6]
Prelude> [3..3]
[3]
```

Types composés : listes

Les types des éléments d'une liste construite par la notation `..` sont limités à des types ordonnés (valeurs ayant un successeur et un prédécesseur). En Haskell, les types préfinis satisfaisant cette condition sont les entiers, les réels et les caractères.

Autres exemples

```
Prelude> [1.4..4.5]
[1.4,2.4,3.4,4.4]
Prelude> [1.4..1.8]
[1.4]
```

Types composés : opérations

Concaténation de listes

L'opérateur ++ prend 2 listes de même type et retourne une nouvelle liste formée des éléments de la première liste suivie des éléments de la seconde.

Exemple

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

Constructeur de listes

Cet opérateur permet de construire une liste à partir d'un élément et d'une liste.

Syntaxe :

<*expression*> `:` <*liste*>

si <*liste*> est de type [*T*], <*expression*> doit aussi être de type *T*.

Exemples :

```
Prelude> 1 : [2,3]
[1,2,3]
Prelude> 1.0 : []
[1.0]
```

- Les opérateurs ++ et `:` ont la même priorité et sont de priorité inférieure aux opérateurs arithmétiques additifs et supérieure aux opérateurs de comparaison (<, >, etc). ++ et `:` ont le niveau de précédence 5.

Types composés : opérations

- Les opérateurs ++ et `:` sont associatifs à droite.

  Exemple

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

  L'évaluation de `1 : 2 : []` se fait comme (1 : (2 : `[]`)).
- Dans une séquence d'opérateurs `:`, l'opérande le plus à droite doit être une liste.
- La définition d'une liste par la notation des parenthèses carrées est en fait un embellissement syntaxique. En interne, tout se passe avec le constructeur `:`.

Sélection

L'opérateur `!!` prend une liste et un entier *i* et retourne le *i*e élément de la liste. La numérotation des éléments de la liste débute par l'élément 0.

```
Prelude> [1,2,3] !! 2
3
Prelude> [1,2,3] !! 3
*** Exception: Prelude.(!!): index too large
```

- L'opérateur `!!` est associatif à gauche et a un niveau de précédence égal à 9 (très prioritaire)

Types composés : chaînes de caractères

Les chaînes de caractères sont des listes de caractères, et les types [*Char*] et *String* sont équivalents. Ainsi, la séquence

```
['H','a','s','k','e','l','l']
```

peut aussi s'écrire

```
'H':'a':'s':'k':'e':'l':'l':[]
```

ou plus simplement

```
"Haskell"
```

Tous les opérateurs et les fonctions valides sur les listes le sont aussi pour des chaînes de caractères.

Exercices

Exercice 1

Donnez les résultats des expressions ci-dessous.

(1) `[1,2,3] !! ([1,2,3] !! 1)`
(2) `head [1,2,3]`
(3) `tail [1,2,3]`
(4) `"a":["b","c"]`
(5) `"abc" ++ "d"`
(6) `tail "abc"  ++ "d"`
(7) `head "abc" : "d"`
(8) `([1,2,3] !! 2 : []) ++ [3,4]`
(9) `[3,2] ++ [1,2,3] !! head [1,2] : []`

Exercice 2

Pourquoi les expressions suivantes contiennent-elles des erreurs?

(1) `head []`
(2) `tail []`
(3) `["n"]:["o","n","!"]`
(4) `1 ++ 2`
(5) `head "abc" ++ "d"`

Exercice 3

Donnez le type des expressions ci-dessous.

(1) `(1.5,("3",[4,5]))`
(2) `[[1,2],[]]`
(3) `(['a','b'],[[],[1,2,3]])`

Exercice 4

Donnez une expression qui satisfasse les types ci-dessous.

(1) `[[[Integer]]]`
(2) `[(Integer, Char)]`
(3) `(Double, [[Integer]])`
(4) `(((Integer, Integer), [Bool], Double), (Double, Char))`

Fonctions

Sous sa forme la plus simple, une fonction prend la forme suivante :

<*signature*> ::= [<*id_var*> {`,`< *id_var*>} `::` <*expression_type*>]

<*définition*> ::= {<*id_var*> {<*gabarit*>}* = <*expression*>}$^{+}$

<*id_var*> ::= <*lettre_minuscule*> {<*lettre_minuscule*> | <*lettre_majuscule*> | <*chiffre10*> | `'`}

La première production désigne le type de la fonction et la seconde production donne sa définition.

Remarques

- La première ligne dans la syntaxe d'une fonction est la *signature de type* de la fonction. Cette ligne n'est pas obligatoire, car Haskell peut inférer par lui-même le type de la fonction.
- La notion de gabarit est une notion plus générale que celle d'un paramètre formel.
- Sans paramètre, une fonction devient une variable. Une variable se comporte comme une fonction retournant une constante.

Exemples simples

```
majuscule c = chr (ord c - 32)
```

La fonction `majuscule` convertit un caractère minuscule en son correspondant majuscule.

```
somme x y = x + y
```

La fonction `somme` prend 2 paramètres et les somme.

Fonctions

Pour utiliser une fonction, il suffit de l'appliquer :

```
> majuscule 'a'
'A' :: Char
> somme 2 3.5
5.5 :: Double
```

Remarques

- Rappelons que l'application de fonctions est plus prioritaire que tous les opérateurs du langage :
  ```
  > somme 2 2*2 -- réalise (2+2)*2 et non 2+(2*2)
  8 :: Integer
  ```
- Notre fonction `majuscule` ne vérifie pas que le caractère passé en paramètre est une lettre et si ce caractère est en minuscule. Une meilleure définition serait
  ```
  majuscule c = if c >= 'a' && c <= 'z' then chr (ord c - 32)
                                        else c
  ```
- Pour inclure la signature des fonctions, on peut écrire
  ```
  majuscule :: Char->Char
  somme :: Integer->Integer->Integer
  ```
  À présent, la fonction `somme` se limite à des entiers alors que précédemment la fonction était polymorphe sur la *classe* `Num`.

Expression de type

<*expression_type*> ::= <*type_parametre*> [`->`<*expression_type*>]

<*type_parametre*> ::= [<*type_parametre*>] <*type_simple*>

<*type_simple*> ::= (<*expression_type*> {`,`<*expression_type*>}$^{+}$) -- tuple

| (<*expression_type*>) -- type entre parenthèses

| [<*expression_type*>] -- type liste

| <*type_var*>

<type_var> ::= `Char` | `Double` | `Float` | `Int` | `Integer` | `String` | <*id_type*>

<*id_type*> ::= <*id_var*>

Remarques

1. L'opérateur `->` est un *constructeur de fonctions*. Si T_1 et T_2 sont des types, T_1`->`T_2 est le type d'une fonction ayant le domaine défini par T_1 et l'image de T_2.
2. L'opérateur `->` est associatif à droite. T_1`->`T_2`->`T_3 s'évalue comme T_1`->(`T_2`->`T_3`)`, c.-à-d. qu'il s'agit d'une fonction ayant le domaine défini par T_1 et retournant une fonction de type T_2`->`T_3.
3. Pour une fonction d'arité supérieure à 1 (c.-à-d. ayant plus d'un paramètre), il y a autant d'interprétations de la fonction qu'elle a d'arité. T_1`->`T_2`->`T_3 peut aussi bien désigner une fonction prenant un paramètre et retournant une fonction de type T_2`->`T_3 ou encore une fonction prenant 2 paramètres et retournant une valeur de type T_3. Ces 2 interprétations sont équivalentes. Cette idée sera revue plus tard quand nous aborderons la notion de *currification de fonctions*.

Expression de type

4. Pour désigner un type quelconque et ainsi avoir des expressions de type polymorphe, il est possible d'utiliser un identificateur là où un type prédéfini serait attendu.

Exemples

1. `[a]->a` est une fonction prenant une liste de type a et retournant une valeur de type a. `[Int]->Int` et `[[Int]]->[Int]` satisfont cette signature.
2. `a->b` est plus général que `a->a`.
3. La signature de la fonction somme ne peut pas s'écrire
   ```
   somme :: a -> a -> a
   ```
   car il est impossible de sommer 2 chaînes de caractères, par exemple. Pour désigner un type quelconque limité à un ensemble d'opérateurs, Haskell introduit la notion de *classes* que nous verrons plus tard. Pour la fonction `somme`, il aurait fallu écrire
   ```
   somme :: Num a => a -> a -> a
   ```
   Si le type a appartient à la classe `Num` — c.-à-d. si les opérations arithmétiques sont définies pour le type a — alors `somme` est de type `a->a->a`.

Exercices

Exercice 1

Écrire une fonction qui retourne le 2e élément d'une liste. La liste contient toujours au moins 2 éléments.

Exercice 2

Écrire une fonction qui retourne le 2e caractère d'une chaîne de caractère. La chaîne contient toujours au moins 2 caractères.

Exercice 3

Écrire une fonction qui réalise une rotation cyclique vers la gauche d'une liste :

si l est `[]` le résultat est `[]`;

si l contient un seul élément, le résultat est la liste l;

si l == [a_1,a_2,...a_n], l'application sur l donne [a_2,a_3,...a_n,a_1].

Exercice 4

Sans utiliser le sélecteur `!!`, écrire une fonction qui prend une liste et qui retourne la liste sans son second élément. La liste contient toujours au moins 2 éléments.

Fonctions récursives

Il est usuel dans les langages fonctionnels de concevoir des fonctions récursivement car les énoncés d'itération sont soit inexistants (langages fonctionnels purs) ou particulièrement peu commode à utiliser (langages fonctionnels impurs).

Une fonction récursive comporte toujours 2 clauses :

1. La *condition d'arrêt* donne les conditions des arguments où le résultat est connu.
2. Le *pas inductif* définit les cas où les arguments de la fonction ne sont traités par la condition d'arrêt mais qu'une définition récursive simplifiant le pas inductif peut être formulée.

Exemple

Écrivons une fonction `inverseListe` qui inverse les éléments d'une liste.

Condition d'arrêt : si *liste* == `[]` alors retourner `[]`.

Pas inductif : si *liste* == *tete* `:` *queue* alors retourner
`inverseListe` *queue* `++ [`*tete*`]`

En Haskell, ceci donne

```
inverseListe liste =
  if liste == [] then []
  else inverseListe (tail liste) ++ [head liste]
```

Exemples d'applications

```
> inverseListe [1,2,3]
[3,2,1] :: [Integer]
> inverseListe ['a','b','c']
"cba" :: [Char]
```

Fonctions mutuellement récursives

Pour écrire une fonction qui appelle une autre et que celle-ci appelle la première, il suffit d'appliquer une fonction avant sa définition.

Exemple

Écrivons une fonction, `elementsImpairs`, qui retourne tous les éléments indicés impairs d'une liste :
`elementsImpairs` $[a_1,a_2,...a_n] = [a_1,a_3,a_5...]$

Cette fonction peut se définir à partir d'une autre, `elementsPairs`, qui retourne tous les éléments indicés pairs d'une liste :
`elementsPairs` $[a_1,a_2,...a_n] = [a_2,a_4,a_6...]$.

Condition d'arrêt : Les deux fonctions retournent `[]` si leur argument, *liste*, est `[]`.

Pas inductif : Si *liste* n'est pas vide,

`elementsImpairs` *liste* =
`[head` *liste*`]` ++ `elementsPairs (tail` *liste*`)`

`elementsPairs` *liste* = `elementsImpairs (tail` *liste*`)`

En Haskel, ceci donne

```
elementsPairs liste =
   if liste == [] then []
   else elementsImpairs (tail liste)
elementsImpairs liste =
   if liste == [] then []
   else [head liste] ++ elementsPairs (tail liste)
```

Exercices

Exercice 1

Écrire une fonction qui réalise i rotations cycliques vers la gauche d'une liste, c.-à-d., si *liste* = $[a_1,a_2,...a_n]$, l'application de cette fonction sur *liste* donne $[a_{i+1},a_{i+2},...a_n,a_1,a_2,...a_i]$.

Exercice 2

Écrire une fonction qui duplique tous les éléments d'une liste. Pour *liste* = $[a_1,a_2,...a_n]$, l'application de cette fonction sur *liste* donne $[a_1,a_1,a_2,a_2,...a_n,a_n]$.

Exercice 3

Comment Haskell déduit-il les types des parametres de la fonction ci-dessous ?

```
exercice3 a b c d =
    if a == b then c+1 else if a > b then c else b+d
```

Exercice 4

Écrire une fonction qui réalise un tri fusion de 2 listes triées. Par exemple, pour $liste_1$ = [1,2,4] et $liste_2$ = [2,5], la fonction retourne [1,2,2,4,5].

Gabarits

Les fonctions nécessitent généralement un traitement qui dépend des valeurs des paramètres effectifs. Au lieu de passer par le biais d'une expression conditionnelle, Haskell, comme tous les autres langages fonctionnels modernes, introduit la notion de *gabarit* (*pattern* en anglais) pour simplifier l'écriture de ces fonctions.

Définition

Un gabarit est la projection de valeurs à un ensemble de paramètres formels.

Exemple

Si le gabarit est *el* : *els* et que la valeur à projeter sur ce gabarit est la liste [1,2,3], *el* assume la valeur 1 et *els* la liste[2,3].

Syntaxe

<*gabarit*> ::= <*id_var*> [+<*entier*> | @ <*gabarit*>] -- paramètre formel
| <*littéral*> -- constante
| _ -- variable anonyme
| (<*gabarit*>) -- gabarit entre parenthèses
| (<*gabarit*> {,<*gabarit*>}$^{+}$) -- tuple
| [<*gabarit*> {,<*gabarit*>}] | [ ] -- liste
| <*gabarit*> : <*gabarit*> -- liste

<*littéral*> ::= <*entier*> | <*réel*> | <*caractère*> | <*chaîne_caractères*> | <*booléen*>

Gabarits : Exemples

forme	exemples	description
<*id_var*>	`liste`	apparie le paramètre formel et le lie à l'expression
	`x`	passée en argument
<*littéral*>	`"chaine"`	apparie l'argument à une constante
	`3.14`	
_		apparie tout; il n'y a aucune liaison
tuple	`(a,b)`	apparie un tuple si ses champs peuvent aussi être
	`(3,b)`	appariés à leur tour
liste	`[1,a,b]`	apparie une liste si ses éléments peuvent aussi
	`[]`	être appariés à leur tour
liste	`el:els`	apparie une liste où `el` devient la tête et `els` la
	`2:[]`	queue de la liste.

Utilisation des gabarits

- Les gabarits ne se limitent pas qu'aux listes et qu'aux tuples ils peuvent aussi s'appliquer à des types définis par l'usager.
- Même si certains gabarits ont un sens, ils ne sont pas forcément admissibles par Haskell :
  `els ++ [el]` n'est pas un gabarit valide.
- Seul l'opérateur arithmétique + peut former un gabarit du moment que la valeur du paramètre formel résultant sera supérieur ou égal à 0 :

```
mystere (x+4) = x
> mystere 5 -- donne 1
> mystere 4 -- donne 0
> mystere 3 -- x doit être >= 4
*** Exception: <interactive>:1: Non-exhaustive
               patterns in function mystere
```

Fonctions avec des gabarits

Chaque cas particulier identifié par l'usage d'un gabarit doit introduire une expression qui est propre au cas selon la syntaxe :
<*définition*> ::= {<*id_var*> {<*gabarit*>}* = <*expression*>}$^{+}$

- Les identificateurs de la fonction doivent être les mêmes.
- Les gabarits doivent s'appliquer à des types identiques.
- Les gabarits sont évalués dans l'ordre qu'elles apparaissent dans la définition de la fonction.
- Contrairement à d'autres langages fonctionnels, il n'est pas nécessaire que la liste des gabarits soit exhaustive.

Exemple

```
inverseListe2 [] = []
inverseListe2 (el:els) = inverseListe2 els ++ [el]
```

Cet exemple se base sur le même raisonnement que la fonction `inverseListe` de la page 35.

À part la simplification d'écriture, `inverseListe2` n'utilise pas d'opérateur d'égalité ou d'inégalité pour lier un argument à ses paramètres formels. La fonction est alors applicable à un ensemble de types plus vaste :

```
> :t inverseListe
inverseListe :: Eq [a] => [a] -> [a]
> :t inverseListe2
inverseListe2 :: [a] -> [a]
```

En utilisant un vocabulaire OO, la concordance se fait par l'application inversée du constructeur. Ainsi,
liste correspond à *el*`:`*els* si et seulement si *liste* = `head` *liste* `: tail` *liste*.

Fonctions avec des gabarits

- Remarquons qu'une liste peut soit être vide ou contenir des éléments. Notre définition de `inverseListe2` traite tous les cas. Si nous omettons l'un ou l'autre de ces cas, une exception est levée si l'application de la fonction ne peut pas concorder les arguments au gabarit restant.

Autres exemples

Exemple 1

```
sommePaires [] = 0
sommePaires ((a,b):ls) = a + b + sommePaires ls
> :t sommePaires
sommePaires :: Num a => [(a,a)] -> a
```

La fonction `sommePaires` réalise la somme des tuples dans une liste.

Exemple 2

```
sommeListes [] = 0
sommeListes ([]:ls) = sommeListes ls
sommeListes ((el:els):ls) = el + sommeListes (els:ls)
> :t sommeListes
sommeListes :: Num a => [[a]] -> a
```

La fonction `sommeListes` réalise la somme de tous les entiers contenus dans une matrice.

Fonctions avec des gabarits

Exemple 3

En mathématique, le nombre de combinaisons de n objets pris par groupes de r objets à la fois est donné par

$$\frac{n!}{(n-r)!\ r!}$$

si $n = r$, la fonction retourne 1 car 0! = 1;

si $r = 0$, la fonction retourne 1.

Le symbole _ peut s'utiliser pour représenter une variable anonyme.

Si nous avons une fonction qui calcule le factoriel d'un entier positif, on peut écrire

```
combinaison _ 0 = 1
combinaison n r =
   if n == r then 1
   else div (factoriel n) (factoriel(n-r)* factoriel r)
```

Il n'est pas possible d'écrire `combinaison` sous la forme

```
combinaison _ 0 = 1
combinaison a a = 1
combinaison n r =
     div (factoriel n) (factoriel (n-r) * factoriel r)
```

car le second gabarit est équivalent à 2 paramètres formels ayant le même identificateur. Hugs nous sort l'erreur

```
Repeated variable "a" in pattern
```

Fonctions avec des gabarits : clause *as*

Rappelons qu'un gabarit peut avoir la forme

<*gabarit*> ::= <*id_var*> @ <*gabarit*>

Cette forme permet de désigner la totalité du gabarit et évite de devoir recomposer les parties identifiées pour manipuler le tout.

Exemple

Réalisons un tri fusion de 2 listes triées par ordre croissant. Pour $liste_1$ = [1,2,4] et $liste_2$ = [2,5], *fusionListes* $liste_1$ $liste_2$ retourne [1,2,2,4,5].

Condition d'arrêt : Si $liste_i$ = `[]` alors retourner $liste_{i \bmod 2 + 1}$.

Pas inductif : Quand les listes $liste_i$ ne sont pas vides, prendre la plus petite valeur des têtes des listes et retourner une liste comportant cet élément suivi de la fusion des 2 listes résultantes :

x_i = `head` $liste_i$

x_1 <= x_2 alors x_1 `: fusionListes (tail` $liste_1$`)` $liste_2$`)`

sinon x_2 `: fusionListes` $liste_1$`(tail` $liste_2$`)`

En Haskell, ceci donne

```
fusionListes liste [] = liste
fusionListes [] liste = liste
fusionListes (l1@el1:els1) (l2@el2:els2) =
   if el1 <= el2 then el1:fusionListes els1 l2
                 else el2:fusionListes l1 els2
```

Fonctions avec des gardes

Forme presque complète des fonctions

<*définition*> ::= {<*id_var*> {<*gabarit*>}*
({| <*garde*> = <*expression*> }$^{+}$ | = <*expression*>)}$^{+}$

<*garde*> ::= <*expression booléenne*> | `otherwise`

Une garde permet d'introduire une clause qui doit être vérifiée pour que l'expression qui lui est associée devienne la valeur retournée par la fonction.

Exemple

```
comparaison x y | x > y = "Premier param est plus petit."
                | x < y = "Second param est plus petit."
                | otherwise = "Les parametres sont égaux."
```

Remarques

- Les gardes sont évaluées dans l'ordre qu'elles apparaissent dans la définition de la fonction.
- Les gardes en dessous de celle qui est vérifiée ne sont pas évaluées.
- Lors de l'exécution, il doit y avoir au moins une garde qui s'évalue à vrai.
- En utilisant conjointement des gabarits avec des gardes, l'évaluation de concordance ne se poursuit pas dès qu'un gabarit est déterminé, même si ses gardes ne sont pas vérifiées.

```
comparaison x y | x > y = "Premier param est plus petit."
                | x < y = "Second param est plus petit."
comparaison _ _ = "Les parametres sont égaux."
```

L'application `comparaison 4 4` provoque une erreur car la première branche est prise et aucune garde est satisfaite.

Fonctions avec des gardes

Autre exemple

L'évaluation séquentielle de haut en bas des gardes simplifie l'écriture des fonctions. Une fonction retournant le maximum de ses 3 paramètres entiers peut s'écrire :

```
maxTrois x y z | x >= y && x >= z = x
               | y >= z = y
               | otherwise = z
```

Si l'évaluation avait été non-déterministe, les 2 gardes après la première auraient dus s'écrire

```
y >= x && y >= z
z >= y && z >= x
```

Exercices

Dans les exercices suivants, utilisez autant de gabarits et de gardes que possible.

Exercice 1

Écrire une fonction qui détermine si une liste passée en paramètre est vide.

Exercice 2

Écrire une fonction qui réalise une rotation cyclique vers la gauche d'une liste :

- si *liste* est `[]` le résultat est `[]`;
- si *liste* contient un seul élément, le résultat est cet élément;
- si *liste* = $[a_1,a_2,...a_n]$, l'application sur *liste* donne $[a_2,a_3,...a_n,a_1]$.

Exercice 3

Écrire une fonction qui duplique tous les éléments d'une liste.

Pour *liste* = $[a_1,a_2,...a_n]$, l'application de cette fonction sur *liste* donne $[a_1,a_1,a_2,a_2,...a_n,a_n]$.

Exercice 4

Écrire une fonction qui résout le calcul récursif d'un entier positif élevé au carré sans utiliser l'opérateur de multiplication.

Condition d'arrêt : $0^2 = 0$

Pas inductif : $n^2 = (n-1)^2 + 2n - 1$

Exercice 5

Écrire une fonction prenant un élément a et une liste *liste* comportant des listes de même type que a. La fonction retourne une nouvelle liste et qui insert a devant chaque liste contenue dans *liste*.

Par exemple, si *liste* = `[[2,3],[],[4]]` et $a = 1$, la liste retournée est

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

Environnement local à une fonction

Lors de la définition d'une fonction, il est utile de pouvoir emmagasiner un calcul intermédiaire ou de définir une fonction locale. La possibilité de définir un environnement local à une fonction accomplit cet objectif et constitue la première forme d'incapsulation. (La seconde est donnée par la notion de *module*.)

Syntaxe

<*environnement_local*> ::= `let`
<*déclaration*> {[`;`]<*déclaration*>}
`in`
<*expression*>

L'étendu des valeurs définies dans la partie déclarative de l'expression `let` peuvent être utilisées dans les déclarations qui suivent et dans l'expression après le mot clé `in`.

Exemple simple

```
puissance8 x =
  let
    carre = x * x;
    puissance4 = carre * carre
  in
    puissance4 * puissance4
```

Environnement local à une fonction

Autre exemple

Réalisons une fonction, `partageListe`, qui prend une liste *liste* en paramètre et retourne un tuple de 2 listes de telle sorte que chaque élément de *liste* se retrouve dans le tuple et que la longueur de *liste* soit égale à la somme des longueurs des listes du tuple.

Condition d'arrêt : Une liste vide donne une paire de listes vides. Une liste comportant un seul élément produit une liste comportant cet élément et une liste vide.

Pas inductif : Pour une liste de 2 ou plus éléments, les 2 premiers éléments de la liste sont attribués à chaque nouvelle liste du tuple.

```
partageListe [] = ([],[])
partageListe [a] = ([a],[])
partageListe (a:b:liste) =
  let
    (l,m) = partageListe liste
  in
    (a:l,b:m)
```

Au lieu de passer par un gabarit pour extraire les 2 liste, l'expression `(l,m) = partageListe liste` aurait pu se faire par la suite des expressions équivalentes :

```
tuple = partageListe liste
l = fst tuple
m = snd tuple
```

Environnement local à une fonction

Autre exemple

À la page 43, nous avons réalisé la fonction `fusionListes`, qui combine 2 listes triées. Pour faire un tri fusion, il suffit de réunir `fusionListes` et `partageListe`.

```
triFusion [] = []
triFusion [a] = [a]
triFusion liste =
  let
    (l,m) = partageListe liste
  in
    fusionListes (triFusion l) (triFusion m)
```

Toute les fonctions utiles au tri peuvent être encapsuler de la manière suivante :

```
triFusion [] = []
triFusion [a] = [a]
triFusion liste =
  let
     fusionListes liste [] = liste
     fusionListes [] liste = liste
     fusionListes l1@(el1:els1) l2@(el2:els2)
        | el1 <= el2 = el1 : fusionListes els1 l2
        | otherwise = el2 : fusionListes l1 els2
     partageListe [] = ([],[])
     partageListe [a] = ([a],[])
     partageListe (a:b:liste) =
        let (l,m) = partageListe liste
        in (a:l,b:m)
     (l,m) = partageListe liste
  in fusionListes (triFusion l) (triFusion m)
```

Environnement local à une fonction

Il existe une seconde façon pour définir un environnement local : la clause `where`.

Forme complète des fonctions

<*définition*> ::= {<*id_var*> {<*gabarit*>}*
({| <*garde*> = <*expression*> }$^{+}$ | = <*expression*>)
[**`where`** <*déclaration*>$^{+}$] }$^{+}$

<*garde*> ::= <*expression booléenne*> | **`otherwise`**

La clause `where` et l'expression `let` se distinguent uniquement par l'étendue de l'environnement local. L'environnement local introduit par une clause `where` englobe toutes les gardes utilisant le même gabarit. L'environnement local introduit par l'expression `let` se limite à l'expression de retour d'une garde.

Exemple 1

La somme de deux nombres élevés au carré peut s'écrire comme

```
sommeCarre m n = carreM + carreN
        where carreM = m * m
              carreN = n * n
```

La clause `where` permet d'introduire des définitions utilisables dans toutes les expressions qui précèdent la clause.

Exemple 2

Le maximum de deux nombres élevés au carré peut s'écrire

```
maxCarre m n | carreM > carreN = carreM
             | otherwise = carreN
             where carreM = m * m
                   carreN = n * n
```

Environnement local à une fonction

ou encore

```
maxCarre m n | carreM > carreN = carreM
             | otherwise = carreN
             where carreM = carre m
                   carreN = carre n
                   carre n = n * n
```

- L'utilité de la clause `where` se voit dans le dernier exemple.
- Les définitions locales peuvent être référées dans les expressions des gardes et aussi dans le résultat de retour.
- Les définitions locales peuvent être référées avant qu'elles sont définies.

Exercices

Exercice 1

Écrire une fonction qui prend une liste d'éléments ordonnés et qui détermine l'élément le plus grand. Par hypothèse, la liste contient au moins un élément.

Exercice 2

Écrire une fonction qui retourne un tuple de 2 éléments contenant la somme des éléments indicés impairs et pairs respectivement. Par exemple, pour [a_1,a_2, a_3, a_4, ...], le tuple (a_1+a_3+..., a_2+a_4+...) est retourné. Par hypothèse, la liste contient au moins 2 éléments.

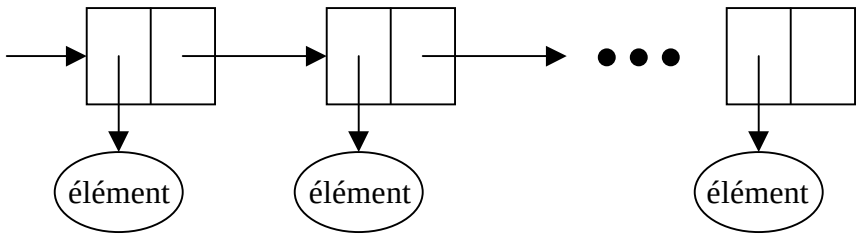
Exercice 3

Écrire une fonction qui calcule, pour un réel $x > 0$ et un entier $i > 0$, la valeur de x^{2^i}. Remarquez que, pour $i = 3$, le calcul est équivalent à $((x^2)^2)^2$.

Traitement efficace des listes

Lors de l'écriture de fonctions faisant intervenir des listes, il est important de comprendre la représentation interne des listes pour réaliser des fonctions plus performantes.

Dans les langages fonctionnelles, une liste se représente par un chaînage simple de ses éléments.



Ainsi, l'opérateur `:` nécessite un temps constant.

Par contre `liste++[element]` est proportionnel à la longueur de `liste`.

Traitement efficace des listes

Analyse de la fonction `inverseListe liste`

```
inverseListe [] = []
inverseListe (a:liste) = inverseListe liste ++ [a]
```

Si $T(n)$ est le temps nécessaire pour inverser une liste de n éléments, nous pouvons trouver une expression pour $T(n)$ en fonction de n.

Condition d'arrêt : Si $n = 0$, vérifier que la liste est `[]` et retourner de l'appel prend un temps constant.

Pas inductif : Pour $n > 0$, la fonction réalise les étapes ci-dessous :

1. Il faut un temps constant pour déterminer que la liste n'est pas vide.
2. Il faut un temps constant pour projeter le paramètre actuel sur le gabarit.
3. Pour inverser la liste, il faut un temps proportionnel à $T(n\text{-}1)$.
4. Pour concaténer le résultat de l'étape 3 à une liste contenant un seul élément. Pour ce faire, il faut traverser la liste et y ajouter un élément.

Pour $n > 0$, notons par

a, le temps nécessaire pour réaliser les étapes 1 et 2.

b, le temps nécessaire pour traverser un élément d'une liste

c, le temps nécessaire pour ajouter un élément à la fin de la liste

d, le temps requis pour retourner à l'appelant.

$$T(n) = a + T(n\text{-}1) + (n\text{-}1)b + c + d$$
$$= n\,(a+c+d) + (n\text{-}1)\,n\,b\,/\,2 + T(0)$$

Ainsi, l'inversion d'une liste de longueur n réalisée par `inverseListe` prend un temps proportionnel à n^2.

Traitement efficace des listes

Pour réaliser une fonction qui inverse une liste en un temps proportionnel à la longueur de la liste, il faudrait toujours ajouter un élément à la tête de la liste.

Pas inductif : si la liste est partiellement inversée, une étape réalise l'opération

$$[a_1,a_2, \ldots a_n][b_1,b_2,b_3, \ldots b_m]$$

$$[a_2, \ldots a_n][a_1,b_1,b_2,b_3, \ldots b_m]$$

```
inverseListe liste =
  let
    invListe [] m = m
    invListe (a:l) m = invListe l (a:m)
  in
    invListe liste []
```

Analyse de `inverseListe`

À chaque pas inductif de `invListe`, il faut un temps constant pour projeter les listes sur leur gabarit, pour ajouter un élément en tête de la liste `m` et pour appeler la fonction récursivement.
À chaque pas, la liste se raccourcit de 1.
Ainsi, nous concluons qu'il faut un temps proportionnel à la longueur de liste pour l'inverser.

Traitement efficace des listes

Exercice 1

Écrire une fonction qui produit la concaténation de 2 listes sans utiliser l'opérateur ++ qui est dédié à cet effet.

Votre fonction devra prendre un temps proportionnel à la longueur de l'une des listes passées en paramètre.

Exercice 2

Écrire une fonction qui réalise i rotations cycliques vers la gauche d'une liste, c.-à-d., si `liste` = [a_1,a_2, … a_n], l'application de cette fonction sur `liste` donne [a_{i+1},a_{i+2}, ... a_n, a_1, a_2, ... a_i].

On supposera que la liste n'est pas vide et que i est inférieur à la longueur de la liste.

Votre fonction devra prendre un temps proportionnel à la longueur de la liste.

Fonctions d'ordre supérieur

- Une fonction pouvant prendre des fonctions comme paramètre et aussi retourner des fonctions comme résultat est dite *d'ordre supérieur*.
- Bien qu'il soit aussi possible de réaliser ceci dans des langages procéduraux, il est possible de prendre et de retourner des *fonctions polymorphes*.

Exemple classique

Calcul d'une intégrale par la méthode des trapèzes.

```
integraleTrapeze inf sup n f =
   if n == 0 || sup-inf <= 0 then 0
   else
      let delta = (sup-inf)/n
      in delta * (f inf + f (inf+delta))/2 +
             integraleTrapeze (inf+delta) sup (n-1) f
```

Le type de la fonction `integraleTrapeze` est

```
integraleTrapeze :: (Ord a, Fractional a) =>
   a -> a -> a -> (a -> a) -> a
```

c.-à-d, que a doit appartenir à la classe `Ord` (pour comparer des valeurs) et aussi à la classe `Fractional` qui étend la classe `Num` avec les opérateurs de division réelle et de réciprocité.

Fonction de projection

Il s'agit d'appliquer une fonction à chaque élément d'une liste.

Définition :

```
projeteFonction f [] = []
projeteFonction f (el:els) =
            f el : projeteFonction f els
```

Remarques

- Le type de la fonction `projeteFonction` est

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

- S'il y a autant de processeur qu'il y a d'éléments dans la liste, `projeteFonction` peut s'exécuter en parallèle (parallélisme SIMD).
- En Haskell, la fonction `projeteFonction` est définie dans *Prelude.hs* et porte le nom `map`.

Exemples

```
> carre a = a * a
> projeteFonction carre [1.0,2.0,3.0]
[1.0,4.0,9.0] :: [Double]
> projeteFonction negate [1.0,-2.0,3.0]
[-1.0,2.0,-3.0] :: [Double]
```

Fonction de réduction

- Il s'agit de réduire une liste d'éléments à une valeur en appliquant une fonction ou un opérateur.
- Généralisation d'un opérateur binaire étendu à une liste.
- Utile en parallèlisme SIMD.

Définition :

```
reduit f a [] = a
reduit f a (el:els) = f el (reduit f a els)
```

Le type de la fonction `reduit` est

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

Exemples

1. La somme des éléments d'une liste devient

   ```
   plus a b = a + b
   reduit plus 0 liste
   ```

   Contrairement aux opérateurs unaires, les opérateurs binaires ne peuvent pas s'utiliser tel quel. Pour convertir un opérateur binaire en une fonction, il suffit de mettre l'opérateur entre parenthèses :

   ```
   > (+) 1 2
   3 :: Integer
   ```

   Ainsi, la somme des éléments d'une liste peut s'écrire

   ```
   reduit (+) 0 liste
   ```

2. Le produit des éléments d'une liste devient

   ```
   produit a b = a * b
   reduit produit 1 liste
   ou
   reduit (*) 1 liste
   ```

Fonction de réduction

3. `reduit (:) [] liste`

   copie une liste

4. `appondre a b = reduit (:) b a`

   réalise la même fonction que l'opérateur ++

Remarques

- La fonction `reduit` s'applique aux opérateurs associatifs où l'ordre de l'évaluation est sans importance : +, *, min, max, et booléen, ou booléan
- La fonction `reduit` est définie dans *Prelude.hs* et s'appelle `foldr`.
- Tel que nous l'avons défini, `reduit` peut accepter une liste vide (c.-à-d. la liste vide est la condition d'arrêt). Il aurait aussi été possible de retourner le dernier élément d'une liste et de lever une exception si la liste est vide.

Fonction de réduction

Exemple pratique

Déterminons la variance des éléments d'une liste.

Rappelons que

$$E[(X - \overline{X})^2] = E[X^2] - E[X]^2$$
$$= \frac{1}{n}\sum x_i^{\,2} - \left(\frac{1}{n}\sum x_i\right)^2$$

```
variance [] = 0
variance liste =
  let donneLongeurListe [] = 0
      donneLongeurListe (el:els) =
                        1 + donneLongeurListe els
      n = donneLongeurListe liste
  in
      reduit (+) 0 (projeteFonction carre liste)
      / n - carre (reduit (+) 0 liste / n )
```

Fonction de filtrage

Ce type de fonction prend un prédicat *p* et une liste *l* en paramètre et produit une nouvelle liste *l'* :

Condition d'arrêt : si *l* = `[]` alors *l'* = `[]` $\forall p$

Pas inductif : *l'* = {*el* | *el* $\in$ *l* et *p el* = vrai}

Définition :

```
filtre p [] = []
filtre p (el:els)
      | p el = el : filtre p els
      | otherwise = filtre p els
```

Remarques

- p est une fonction `a->Bool` et la liste contient des éléments de type a.
- Le résultat est une nouvelle liste de type a.
- Le type de la fonction `filtre` est alors `(a -> Bool) -> [a] -> [a]`
- La fonction `filtre` est définie dans Prelude.hs et s'appelle `filter`.

Exemple

```
> plusGrandQueHuit x = x > 8
> filtre plusGrandQueHuit [1,15,5,10]
[15,10] :: [Integer]
```

Exercices

Exercice 1

Réalisez une fonction qui incrémente tous les éléments d'une liste d'entiers.

Exercice 2

Réalisez une fonction qui remplace tous les nombres < 0 par 0 dans une liste. Les éléments > 0 demeurent inchangés.

Exercice 3

Réalisez une fonction qui retourne la plus grande valeur d'une liste d'entiers. La valeur 0 est retournée si la liste est vide.

Exercice 4

Réalisez une fonction qui prend une liste formée de chaînes de caractères et qui retourne la même liste mais ayant ses éléments ne dépassant pas 5 caractères de long. Les chaînes plus longues que 5 sont tronquées.

Exercice 5

La performance de la fonction `integraleTrapeze` peut s'améliorer si l'intervalle `delta` se calcule une seule fois et si le nombre d'évaluation de la fonction `f` est minimisé. Réécrire `integraleTrapeze` en tenant compte de cette suggestion.

Ensembles ZF (Zermelo-Fränkel)

Il est fréquent en math de définir un ensemble de valeurs en donnant les conditions que doivent satisfaire ces valeurs.

Par exemple,

$\{x^2 \mid x \in N \wedge impair(x)\}$

Avec cette formulation, 3 composants interviennent dans la définition de l'ensemble :

1. un générateur fournissant les éléments définis ($x \in N$).
2. un filtre qui est un prédicat déterminant, des valeurs générées, celles qui seront utilisées pour former l'ensemble ($impair(x)$).
3. une expression permettant de manipuler les valeurs générées et satisfaisant les prédicats avant de les placer dans l'ensemble.

En Haskell, l'exemple précédant (sans la fonction *impair*) s'écrit

```
[x*x | x <- [1..], impair x]
```

Syntaxe

<*ensemble ZF*> ::= [<*expression*> | <*qualificateur*>{,<*qualificateur*>}*]

<*qualificateur*> ::= <*gabarit*> <- <*expression*> -- générateur

| <*expression*> -- prédicat retournant un résultat de type Bool

Ensembles ZF (Zermelo-Fränkel)

Remarques

- La suite de symboles `<-` représente le symbole $\in$.
- La virgule séparant les qualificateurs est interprétée comme $\wedge$.
- Quand l'ensemble ZF utilise plusieurs générateurs, ces générateurs produisent des valeurs dont le générateur le plus à droit varie en premier.
  `[(x,y) | x <- [1,2], y <- [3,4]]`
  donne l'ensemble
  [(1,3),(1,4),(2,3),(2,4)]

Exemples

```
> impair x = mod x 2 == 1
> [impair x | x <- [1..3]]
> [True,False,True] :: [Bool]

> additionnePairs liste = [m + n | (m,n) <- liste]
> additionnePairs [(1,2),(5,3),(3,5)]
> [3,8,8] :: [Integer]

> additionnePairsOrdonnees liste =
                                  [m + n | (m,n) <- liste, m < n]
> additionnePairsOrdonnees [(1,2),(5,3),(3,5)]
> [3,8] :: [Integer]

> triangle n = [(x,y) | x <- [1..n], y <- [1..x]]
> triangle 3
> [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)] :: [(Integer,Integer)]

> repete n = [x | x <- [1..n], y <- [1..x]]
> repete 3
> [1,2,2,3,3,3] :: [Integer]
```

Les fonctions *projeteFonction* et *filtre* peuvent se définir élégamment avec des ensembles ZF :

```
projeteFonction f liste = [f el | el <- liste]
filtre p liste = [el | el <- liste, p el]
```

Exercices

Exercice 1

Quels sont les ensembles ZF définis par les expressions suivantes ?

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

(b) `[(x,y) | x <- [1..4], mod x 2 == 0, y <- "ab"]`

Exercice 2

Définissez l'ensemble ZF donnant les listes ci-dessous.

(a) [1,2,3,4,5,6,7,8,10,11,12,13]

(b) [2,-3,4,-5,6,-7,8,-9,10,-11,12,-13]

Exercice 3

En utilisant les ensembles ZF, réalisez une implémentation de l'algorithme du tri rapide qui tri une liste.

Instantiation partielle de fonctions

Quand la définition d'une fonction est un identificateur suivi par une liste de paramètres sans parenthèse ni virgule, la forme obtenue est dite *de Curry*.

Dans ce qui suit, nous dirons qu'une fonction définie ainsi est une fonction *curryfiée*.

Exemple

```
> curryExponentielle _ 0 = 1
  curryExponentielle x y =
                        x * curryExponentielle x (y-1)
curryExponentielle :: (Num a, Num b) => a -> b -> a
> curryExponentielle 4.0 3
64.0 :: Double
```

Remarques

- `curryExponentielle` est de type
  `(Num a, Num b) => a -> b -> a`,
  c.-à-d. `(Num a, Num b) => a -> (b -> a)`.
- Dans l'application `curryExponentielle 4.0 3`, `curryExponentielle` prend le premier argument (4.0) et retourne une nouvelle fonction. Cette dernière est de type Integer $\rightarrow$ Double et elle prend un entier *y* comme argument pour produire 4.0^{y}.

La forme *curryfiée* d'une fonction est utile par le fait qu'elle permet de construire de nouvelles fonctions à partir d'une fonction en liant certains mais pas tous les paramètres de cette fonction à ses arguments.

Instantiation partielle de fonctions

La liaison partielle des paramètres formels d'une fonction à des arguments revient à *instantier partiellement* cette fonction.

Exemple

```
> exponentielle4 = curryExponentielle 4.0
exponentielle4 :: Integer -> Double
```

Remarques

- `exponentielle4` retourne le résultat de 4.0 élevé à un exposant entier.
- `exponentielle4` n'est pas une nouvelle définition de fonction, mais une instantiation particulière d'une fonction existante.
- Pour utiliser `exponentielle4` , il suffit de lui fournir un entier comme argument :

  ```
  > exponentielle4 3
  64.0 :: Double
  ```

- Pour instancier partiellement une fonction curryfiée sans suivre l'ordre dans lequel ses paramètres apparaissent, il faut passer par des paramètres supplémentaires :

  ```
  > carre x = curryExponentielle x 2
  carre :: Num a => a -> a
  ```

Exercices

Exercice 1

Écrire une fonction currifiée, `appliqueListe`, prenant une liste de fonctions et une valeur et qui applique chaque fonction à la valeur passée comme paramètre.

Exercice 2

1. Écrire une fonction qui détermine si une sous-chaîne est contenue dans une chaîne de caractères. Il est préférable de currifier cette fonction.

   En utilisant `projeteFonction`, réaliser une liste de fonctions à partir d'une liste de chaînes de caractères [s_1,s_2,...s_n] de telle sorte qu'une fonction f_i de la liste vérifie si la sous-chaîne s_i est contenue dans une chaîne de caractères.
2. Instancier cette fonction avec la liste `["a","bout","a bout"]`.
3. Que donne `appliqueListe` avec la liste obtenue et la chaîne `"aboutir"`?

Exercice 3

Écrire une fonction `curryfie` qui convertit une fonction prenant un tuple de 2 champs dans sa forme curryfiée.

Exercice 4

Écrire une fonction `decurryfie` qui convertit une fonction définie sous une forme curryfiée dans une forme prenant un tuple. Le nombre de paramètres de la fonction à décurryfier est 2.

Réponses

Exercice 1

```
appliqueListe [] x = []
appliqueListe (el:els) x = el x : appliqueListe els x
```

Exercice 2

```
-- estSousChaine determine si tous les elements de la premiere
-- liste sont contenus dans la seconde liste.
estSousChaine [] _ = True
estSousChaine _ [] = False
estSousChaine l  m@(el:els)
   | estPrefixeListe l m = True
   | otherwise = estSousChaine l els
     where
        -- estPrefixeListe determine si tous les elements de la
        -- premiere liste sont identiques aux premiers elements
        -- de la seconde liste.
        estPrefixeListe [] _ = True
        estPrefixeListe _ [] = False
        estPrefixeListe (sel:sels) (el:els)
              | sel == el = estPrefixeListe sels els
              | otherwise = False
> sontSousChaines = projeteFonction estSousChaine ["a","bout",
    "a bout"]
sontSousChaines :: [[Char] -> Bool]
> appliqueListe sontSousChaines "aboutir"
[True,True,False] :: [Bool]
```

ou encore

```
> appliqueListe (projeteFonction estSousChaine ["a","bout","a
    bout"]) "aboutir"
[True,True,False] :: [Bool]
```

Réponses

Exercice 3

```
> curryfie f x y = f(x,y)
curryfie :: ((a,b) -> c) -> a -> b -> c
```

Exercice 4

```
> decurryfie f(x,y) = f x y
decurryfie :: (a -> b -> c) -> (a,b) -> c
```

Évaluation paresseuse

Seuls les arguments nécessaires au calcul d'une fonction sont évalués. Si l'argument est un type composé (liste ou tuple), seules les parties nécessaires sont examinées.

Exemple

```
f x y = x + y
```

Pour l'évaluation de `f (9-3) (f 3 4)`, les 2 arguments ne sont pas évalués avant l'appel de `f`; c'est uniquement lors de l'évaluation de l'opérateur + qu'il y a l'évaluation des arguments.

```
f (9-3) (f 3 4)
⇒ (9 - 3) + (f 3 4)
⇒ 6 + (f 3 4)
⇒ 6 + (3 + 4)
⇒ 6 + 7
⇒ 13
```

Autre exemple

```
g x y = x + 20
```

L'appel `g (9-3) (g 3 4)` donne

```
⇒ (9 - 3) + 20
⇒ 6 + 20
⇒ 26
```

Évaluation paresseuse

Un argument dupliqué n'est jamais évalué plus d'une fois. En l'interne, il y a un graphe pour représenter les expressions.

Exemple

```
h x y = x + x
h (9-3) (h 3 4)
⇒ (9 - 3) + (9 - 3)
⇒ 6 + 6
⇒ 12
```

En présence de gabarits, les arguments sont évalués *suffisamment* pour établir une correspondance.

Exemple

```
f [] _ = 0
f _ [] = 0
f (x:xs) (y:ys) = x + y
```

L'évaluation de `f [1..] [3..]` donne

```
f [1..] [3..]
⇒ f (1:[2..]) [3..]        -- vérification de []
⇒ f (1:[2..]) (3:[4..])    -- vérification de []
⇒ 1 + 3
⇒ 4
```

Évaluation paresseuse

En présence de gardes, l'évaluation débute à la première garde et continue de gardes en gardes jusqu'à ce qu'une garde est vraie, puis l'évaluation se poursuit avec la clause associée.

Exemple

```
f m n p | m >= n && m >= p = m
        | n >= m && n >= p = n
        | otherwise = p
```

L'évaluation de `f (3+2) (4*8) (3+9)` donne

```
f (3+2) (4*8) (3+9)
⇒ m >= n && m >= p
  ⇒ (3+2) >= (4*8) && (3+2) >= (3+9)
  ⇒ 5 >= (4*8) && 5 >= (3+9)
  ⇒ 5 >= 32 && 5 >= (3+9)
  ⇒ False && 5 >= (3+9)
  ⇒ False
⇒ n >= m && n >= p
  ⇒ 32 >= 5 && 32 >= (3+9)
  ⇒ True && 32 >= (3+9)
  ⇒ 32 >= (3+9)
  ⇒ 32 >= 12
  ⇒ True
⇒ n
  ⇒ 32
```

Exercice

Donnez toutes les étapes de l'évaluation de

```
reduit (+) 0 (projeteFonction (^3) [1..2])
```

Listes infinies

L'évaluation de la fonction

```
uns = 1 : uns
```

donne une liste infinie. Les listes infinies `[m ..]` et `[m, n ..]` sont définies en interne comme des fonctions :

```
listeIncrement m = m : listeIncrement (m+1)
listePas m pas = m : listePas (m+pas) pas
```

Si nous définissons

```
additionDeuxElements (x:y:zs) = x + y
```

L'appel `additionDeuxElements uns` s'évalue comme suit

```
additionDeuxElements uns
⇒ additionDeuxElements (1:uns)
⇒ additionDeuxElements (1:1:uns)
⇒ 1 + 1
⇒ 2
```

L'usage de listes infinies nous donne une nouvelle manière de décomposer un problème : la génération des valeurs se sépare de la transformation ou de la sélection des valeurs générées.

Listes infinies

Exemple

Pour déterminer si un nombre est premier, une façon de procéder est de générer une liste de nombres premiers et de déterminer si le nombre recherché s'y trouve dans la liste.

```
premiers = erathosthenes [2..]
erathosthenes (x:xs) =
          x : erathosthenes [y | y <- xs, mod y x > 0]
membreOrdonne (x:xs) n
              | x < n = membreOrdonne xs n
              | x == n = "est permier"
              | otherwise = "n'est pas premier"
estPremier n = membreOrdonne premiers n
```

Listes infinies

L'évaluation de `estPremier 5` donne

```
estPremier 5
⇒ membreOrdonne premiers 5
⇒ membreOrdonne (erathosthenes [2..]) 5
⇒ membreOrdonne (erathosthenes (2:[3..]) 5
⇒ membreOrdonne (2:erathosthenes [y|y <- [3..], mod y 2 > 0]) 5
  ⇒ 2 < 5
  ⇒ True
  ⇒ membreOrdonne (erathosthenes [y|y <- [3..], mod y 2 > 0]) 5
  ⇒ membreOrdonne (erathosthenes (3:[y|y<-[4..],mod y 2 > 0]) 5
  ⇒ membreOrdonne (3:erathosthenes [z | z <- [y | y <- [4..],
                                mod y 2 > 0], mod z 3 > 0]) 5
    ⇒ 3 < 5
    ⇒ True
    ⇒ membreOrdonne (erathosthenes [z | z <- [y | y <- [4..],
                                 mod y 2 > 0],mod z 3 > 0]) 5
    ⇒ membreOrdonne (erathosthenes (5:[z | z <- [y | y <-
                          [6..], mod y 2 > 0], mod z 3 > 0])) 5
    ⇒ membreOrdonne (5:erathosthenes [y | y <- [z | z <- [y |
        y <- [6..],mod y 2 > 0], mod z 3 > 0], mod y 5 > 0]) 5
      ⇒ 5 < 5
      ⇒ False
      ⇒ 5 == 5
      ⇒ True
      ⇒ "est premier"
```

Listes infinies

L'évaluation de `estPremier 4` donne

```
estPremier 4
⇒ membreOrdonne premiers 4
⇒ membreOrdonne (erathosthenes [2..]) 4
⇒ membreOrdonne (erathosthenes (2:[3..]) 4
⇒ membreOrdonne (2:erathosthenes [y|y <- [3..], mod y 2 > 0]) 4
  ⇒ 2 < 4
  ⇒ True
  ⇒ membreOrdonne (erathosthenes [y|y <- [3..], mod y 2 > 0]) 4
  ⇒ membreOrdonne (erathosthenes (3:[y | y <- [4..],
                                      mod y 2 > 0]) 4
  ⇒ membreOrdonne (3:erathosthenes [z | z <- [y | y <- [4..],
                              mod y 2 > 0], mod z 3 > 0]) 4
    ⇒ 3 < 4
    ⇒ True
    ⇒ membreOrdonne (erathosthenes [z | z <-
          [y | y <- [4..], mod y 2 > 0], mod z 3 > 0]) 4
    ⇒ membreOrdonne (erathosthenes (5:[z | z <- [y | y <-
          [6..], mod y 2 > 0], mod z 3 > 0])) 4
    ⇒ membreOrdonne (5:erathosthenes [y | y <- [z | z <-
      [y|y <- [6..], mod y 2 > 0], mod z 3 > 0], mod y 5 > 0]) 4
      ⇒ 5 < 4
      ⇒ False
      ⇒ 5 == 4
      ⇒ False
      ⇒ True
      ⇒ "n'est pas premier"
```

Exercices

Exercice 1

Réalisez une fonction qui retourne la suite infinie des nombres

(a) factoriels [0!,1!,2!,..]

(b) de Fibonacci [0,1,1,2,3,5,8,..]

Exercice 2

Réalisez une fonction qui retourne la somme des termes

[a_0, $a_0 + a_1$, $a_0 + a_1 + a_2$,..] à partir de la liste infinie [a_0, a_1, a_2,..].

Exercice 3

Déterminer la racine carrée d'un nombre x par une approximation successive de nombres provenant de l'équation de récurrence de Newton-Raphson :

$a_0 = x / 2$

$a_n = (a_{n-1} + x / a_{n-1}) / 2$

(a) Réalisez la fonction `newtonRaphson` qui retourne la suite infinie des valeurs [a_0, a_1, a_2, ..] :

```
newtonRaphson :: double -> [double]
```

(b) Réalisez la fonction `precision` qui retourne la première valeur a_n satisfaisant $|a_n - a_{n-1}| < \varepsilon$ quand les a_i proviennent d'une liste et que ε est un paramètre :

```
precision :: double -> [double] -> double
```

Pour trouver la racine carrée qu'un nombre, on devrait pouvoir faire

```
precision epsilon (newtonRaphson x)
```