Série d'exercices #2

IFT-2035

May 5, 2025

2.1 Grammaire micro-JS

Voici un exemple de programme dans un langage similaire à Javascript:

```
var ackermann = function (m, n) {
  if (m == 0) {
    return n + 1;
  } else if (n == 0) {
    return ackermann (m - 1, 1);
  } else {
    return ackermann (m - 1, ackermann (m , n - 1));
  }
};
print (ackermann (2, 2));
```

Voici une partie de la grammaire du langage en notation EBNF:

```
<program> ::= <instr>

<instr> ::= <expr> ";"
          | <instr> <instr>

<expr> ::= "function" "(" <formals> ")" "{" <instr> "}"
         | <expr> "==" <expr>
         | <identifier> "(" <actuals> ")"

<formals> ::= ϵ
            | <identifier> { "," <identifier> }

<actuals> ::= ϵ | <expr>
```

Compléter la grammaire pour qu'elle accepte le programme.

2.2 Conversion de base

Écrire les fonctions suivantes en Haskell pour convertir des nombres en représentation binaire à décimal et vice versa (à remarquer que l'argument est un entier et que par exemple l'entier "cent-un" représente le nombre binaire "un-zéro-un" qui correspond à 5 en décimal). Vous aurez besoin des fonctions prédéfinies mod, et div.

$$\textit{bin2dec}\ 10001 \quad \rightsquigarrow^* \quad 17$$
$$\textit{dec2bin}\ 17 \quad \rightsquigarrow^* \quad 10001$$

Écrire la fonction *baseconv* en Haskell qui convertit d'une base à une autre (≤ 10). N'hésitez pas à définir des fonctions auxiliaires si nécessaire.

$$\textit{baseconv}\ 2\ 10\ 10001 \quad \rightsquigarrow^* \quad 17$$
$$\textit{baseconv}\ 10\ 2\ 17 \quad \rightsquigarrow^* \quad 10001$$

Donner aussi le type de chacune des fonctions que vous avez définies.

Note: La distinction entre un objet et sa représentation est un thème qui réapparait souvent dans ce cours, par exemple sous la forme de la différence entre la syntaxe et la sémantique des programmes.

2.3 Manipulation de listes

Implanter en Haskell les fonctions suivantes:

1. *somme* :: [Int] → Int
   Prend une liste d'entiers et renvoie la somme de ses éléments.

2. *ajoute1s* :: [Int] → [Int]
   Prend une liste d'entiers et renvoie une liste contenant les mêmes entiers auxquels on a ajouté 1.

3. *enleve* :: Int → [Int] → [Int]
   Enlève de la liste d'entiers tous les éléments qui sont plus petits ou égaux au premier argument de la fonction.

4. *retourne* :: [Int] → [Int]
   Prend une liste d'entiers et renvoie une liste contenant les mêmes entiers mais dans l'ordre inverse.

2.4 Quicksort

Implanter en Haskell une variante de *quicksort* pour des listes d'entiers. En clair, trier une liste comme suit:

1. choisir un élément, que l'on nommera le pivot.

2. partitionner la liste en deux sous-listes d'éléments plus petits resp. plus grands que le pivot.

3. trier les deux sous-listes.

4. combiner ces sous-listes triées et le pivot en une liste triée.

Le type sera: *quicksort* :: [Int] → [Int].
Il faudra peut-être définir une ou plusieurs fonctions auxiliaires.
L'opération de concaténation de deux listes s'écrit ++ en Haskell:

$$[1,2] \mathbin{++} [4,5,6] \quad \equiv \quad (++)\ [1,2]\ [4,5,6] \quad \rightsquigarrow^* \quad [1,2,4,5,6]$$

Finalement, généraliser la fonction de tri précédente pour pouvoir l'appliquer à des listes quelconques (pas seulement [Int]), en passant un argument supplémentaire qui indique l'opération de comparaison à utiliser.

Donner aussi le type de cette fonction plus générale et de toutes les fonctions auxiliaires que vous avez définies.