Série d'exercices #2

IFT-2035

September 9, 2024

2.1 Si et seulement si

Voici une grammaire pour if... then... else. Les éléments E et X représentent des expressions et parties de la grammaire qu'il n'est pas important de spécifier ici.

$$\begin{array}{rcl} S & ::= & X \\ & | & \text{“if”}\ E\ \text{“then”}\ S \\ & | & \text{“if”}\ E\ \text{“then”}\ S\ \text{“else”}\ S \end{array}$$

Cette grammaire est ambiguë.

1. Donner un exemple d'ambiguïté.
2. Donner une grammaire non ambiguë qui associe les else avec le if le "plus proche", comme le font les langages de programmation habituels.

2.2 Ambiguïté et récursion

Soit la grammaire suivante pour des expressions arithméthiques

$$\begin{array}{rcl} \mathit{expr} & ::= & \mathit{expr} + \mathit{expr} \\ & | & \mathit{expr} * \mathit{expr} \\ & | & \mathit{number} \end{array}$$

1. Montrer que cette grammaire est *ambiguë*.
2. Réécrire cette grammaire de manière à éliminer les ambiguïtés.
3. Cette grammaire est *récursive à gauche*, ce qui pose problème pour certaines techniques d'analyse syntaxique: En effet, dans un parseur avec analyse descendante (top down), la portion du programme devant lire la catégorie *expr* va devoir d'abord faire appel à la portion du programme qui doit lire la catégorie *expr* ...
   Réécrire la grammaire de manière à éviter cette *récursion à gauche*.

2.3 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.

2.4 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.