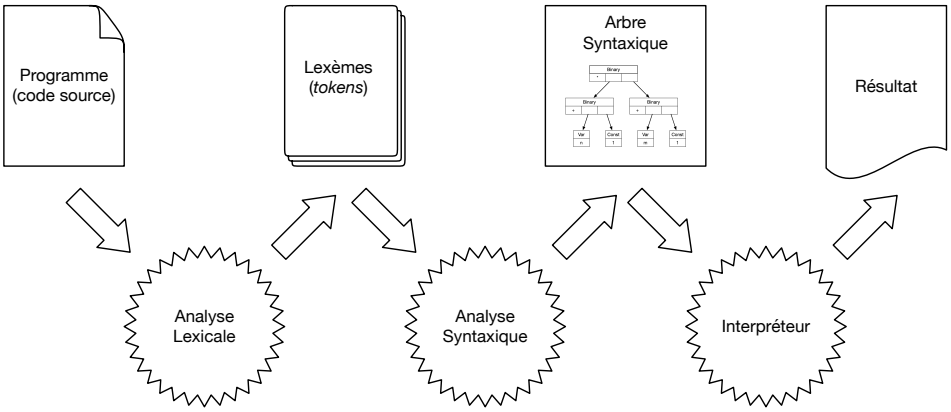
Programmation Fonctionnelle Avancée
Laboratoire 6

Interpréteur

Le but de ce laboratoire consiste à jeter les bases d'un interpréteur de langage de programmation simple.

Normalement un interpréteur est construit sur le modèle suivant :



Le programme source passe à travers une phase d'analyse lexicale qui permet de découper ce programme en unités lexicales (*tokens*), puis à travers une analyse syntaxique qui construit un *AST* (*abstract syntax tree*). Cet arbre peut alors être analysé (par exemple pour inférer les types des expressions) puis finalement évalué pour donner un résultat. C'est ce qui se passe lorsqu'on soumet une expression à l'interpréteur Haskell.

Ce laboratoire se base sur l'embryon de déclaration suivant :

```
data Expr = Const Int | Sum Expr Expr | Prod Expr Expr | ...
```

1. Le but de ce laboratoire est d'étendre la déclaration de type ci-dessus pour inclure d'autres opérateurs unaires et binaires ainsi que des expressions conditionnelles.

On écrira alors une fonction eval pour évaluer le résultat d'expressions complexes.

Attention : Afin de ne pas vous perdre dans les complications on admet pour ce laboratoire que le résultat d'une expression est toujours un nombre entier. Les expressions conditionnelles fonctionneront donc *à la C* ou 0 est considéré comme *faux* et tout autre valeur comme *vrai*.

2. Dans un second temps, redéfinissez la fonction **`show`** en dérivant explicitement le type **`Expr`** de la classe **`Show`** afin d'afficher des expression de manière lisible.

Vous êtes libres d'adopter le formatage qui vous convient.

3. Finalement, introduisez la possibilité de définir les variables locales qui peuvent être utilisées par leur nom, une fois leur valeur définie par une expression.

On se base donc sur le modèle Haskell :

```
let var = expr1 in expr2
```

La valeur de la variable **`var`** est définie par l'expression **`expr1`**, elle peut être utilisée par son nom dans **`expr2`**.

Attention : Il faudra compléter la notion d'expression par deux nouvelles alternatives : une pour définir une variable, par exemple **`Let String Expr Expr`** et une autre pour l'utilisation de cette variable.

Bien entendu il faudra compléter la fonction **`eval`** afin d'avoir un second paramètre, un environnement, qui donne la valeur des variables. Cet environnement peut par exemple être une liste de paires variable-valeur.

Rendu

Sont attendus pour le rendu :
- Un rapport (une à deux pages A4) qui explique la solution proposée et la documente.
- Le code source sous forme d'un module général (qui peut inclure des modules secondaires).
- Des exemples d'exécution.

L'implémentation doit définir au moins :
- Deux opérateur unaires.
- Six opérateurs binaires.
- Un énoncé d'expression conditionnelle.
- Un énoncé permettant de définir et utiliser une variable.
- La fonction **`show`** sur le type **`Expr`**.

Notation
- Un point pour les opérateurs.
- Un point pour la fonction **`show`**.
- Un point pour l'énoncé conditionnel.
- Un point pour l'a définition et l'utilisation de variable.
- Un point pour le rapport.