Projet de Programmation Fonctionnelle

Enseignants : Mme. Agnès Arnould & M. Patrice Naudin
Réalisé par : Vincent Commin & Louis Leenart

Introduction

Ce projet est à réaliser avant le 9 avril 2021, dans le cadre de l'Unité d'Enseignement "Programmation Fonctionnelle". Ce projet porte sur la simplification d'expressions arithmétiques que nous pouvons diviser en 3 parties :

- Analyse syntaxique et construction de l'arbre
- Simplification de l'arbre
- Affichage du résultat

Le détail des contraintes imposées est contenu dans le fichier `sujet.pdf` contenu dans la racine. L'ensemble des fichiers sources sont contenus dans le dossier `src`.

Analyse syntaxique et contruction de l'arbre

Pour la construction de l'arbre, nous nous sommes servis de la notation polonaise inversé (notation postfixe). Celle-ci permet de directement avoir les opérations prioritaires par rapport aux autres. Ainsi `13 2 5 * 1 0 / - +` est la notation postfixée de `13 + 2 * 5 - 1 / 0`.

Les fonctions prenant en entrée une équation en tant que chaine de caractères et retournant sa version postfixée nous étant déjà fourni, nous avons pu nous concentrer sur le parser `parse(token list) : tree`. Pour chaque nombre de la chaine, il nous suffit de les empiler en tant que type `tree`, et de les dépiler en enracinant l'arbre à chaque opération (addition, soustraction, multiplication, division et le moins).

Simplification de l'arbre

Pour simplifier l'expression arithmétique créée à partir de l'AST, nous avons appliqué un pattern matching pour les opérations de simplification basiques. En effet, la simplification de `x * 0`, `x + 1` ou `x - x` sont évidentes. La fonction utilisée pour simplifier les-dites expressions est `simplify(tree) : tree`, qui nécessite donc l'arbre à simplifier en entrée et la sortie correspond à l'arbre simplifié.

On note que nous n'avons pas mis en place toutes les opérations de simplifications possible, cependant uniquement avec les cas basiques que nous avons implémenté, nous obtenons des résultats concluants. La liste des opérations que nous simplifions est la suivante :

- `x * 1 | 1 * x -> x`
- `x + 0 | 0 + x -> x`
- `x * 0 | 0 * x -> 0`
- `x - x -> 0`
- `x / x -> 1`
- Evaluation de sous-arbre composé uniquement de constantes

Affichage du résultat

Pour l'affichage de l'arbre, il nous suffit de parcourir l'arbre en mode infixe et d'afficher chaque noeud. Cependant, nous avons rencontré un problème. Si nous voulons afficher l'équation avec la bonne priorité pour les opérateurs, il faudrait sur-parenthéser l'équation ce qui devient très vite indigeste. C'est pour cela qu'à chaque fois que nous allons afficher un opérateur, on vérifie si celui-ci est prioritaire par rapport à ses fils. S'il est prioritaire, alors il faudra afficher le fils entre parenthèses.

Utilisation de notre module

Lancement

Pour lancer le programme, il suffit de lancer l'exécutable `ast` puis en entrant l'expression en notation postfixée dans la console, ou alors en effectuant une redirection.

```
./ast < input
```

On note que l'entrée doit contenir l'expression postfixée et doit se terminer par un **";"**.
Par exemple, pour l'entrée `x 3 + 5 7 + + 3 4 * 1 3 + / /;`, le résultat est :

```
Expression: (x+3+5+7)/(3*4)/(1+3)
Après simplification: (x+3+12)/3
```

Compilation

La compilation de notre exécutable se fait via le fichier `compile.sh` présent dans le dossier `src`. Lancez alors ce fichier avec la commande suivante.

```
bash compile.sh
```

Si vous utilisez un programme autre, ou que vous rencontrez des problèmes, nous utilisons la commande suivante pour compiler.

```
ocamlc expression_scanner.cmo ast.ml -o ast
```

La version d'Ocaml utilisée est `Ocaml 4.08.1`, si la votre est différente, remplacer le module `expression_scanner` par la version appropriée.