

# DM COMP : Création d'un parseur d'expressions arithmétiques

Aurèle Barrière & Antonin Garret

29 septembre 2016

## 1 Introduction

Le but de ce projet est d'implémenter un parseur d'expressions arithmétiques en **OCaml**, en utilisant les **stream parsers** de l'extension **Camlp4**. Le but est ainsi de convertir une chaîne de caractères (écrite sur un alphabet comportant des nombres et des opérateurs arithmétiques) en un arbre syntaxique abstrait donnant une représentation correcte (c'est-à-dire qui respecte les propriétés de priorité et d'associativité voulues) de l'expression.

On effectuera une analyse descendante.

Dans une première partie, nous verrons comment aboutir à une grammaire d'expressions arithmétiques prête à être implémentée. Ensuite, nous verrons comment l'implémenter. Enfin, nous nous intéresserons à la vérification empirique de notre implémentation.

## 2 Une grammaire pour des expressions arithmétiques

Pour implémenter notre parseur avec analyse descendante avec les **stream parsers**, il faut définir un ensemble de fonctions mutuellement récursives qui décomposent la chaîne de caractère souhaitée. Dans un premier temps, la chaîne de caractère aura été parcourue par un *scanner* (ou **lexer**) qui repère les mot-clés (ou **tokens**), et qui convertit donc une chaîne de caractères en une chaîne de mot-clés.

Les fonctions mutuellement récursives prennent donc en entrée une chaîne de mot-clés, et renvoient un arbre syntaxique abstrait d'expression, qu'il nous faut définir. Une expression sera ainsi : un nombre ou une addition ou soustraction ou multiplication ou division ou puissance de deux expressions. On aura donc en **OCaml** :

```
type expr = Num of int | Add of expr * expr | Sub of expr * expr | Mul
           of expr * expr | Div of expr * expr | Pow of expr * expr
```

### 2.1 Une première version

Pour parser correctement des expressions arithmétiques dans lesquelles figurent l'addition, la soustraction, la multiplication et des parenthèses, en respectant les propriétés suivantes :

- L'addition, la soustraction et la multiplication sont associatives à gauche.
- Les parenthèses sont prioritaires, puis la multiplication, puis l'addition (ou soustraction).

On dispose d'une première grammaire. Les récursivités gauches témoignent de l'associativité gauche des opérateurs. On a décomposé nos expressions en plusieurs sous-expressions : (E : expressions, F : facteurs, A : atomes) en suivant l'ordre inverse de priorité. Les dépendances cycliques entre ces différentes fonctions récursives nous obligeront à les déclarer comme des fonctions mutuellement récursives.

$$\begin{aligned} E &\rightarrow E + F \mid E - F \mid F \\ F &\rightarrow F * A \mid A \\ A &\rightarrow \text{constante} \mid ( E ) \end{aligned}$$

## 2.2 Ajout de la division

La division est un opérateur qui doit être associatif à gauche et avoir la même priorité que la multiplication. On ajoute donc la règle  $F \rightarrow F / A$ .

$$\begin{aligned} E &\rightarrow E + F \mid E - F \mid F \\ F &\rightarrow F * A \mid F / A \mid A \\ A &\rightarrow \text{constante} \mid ( E ) \end{aligned}$$

## 2.3 Élimination des récursivités gauches

Cependant, la méthode utilisée est l'analyse descendante. On ne peut donc pas garder nos règles avec récursivité gauche pour garantir la terminaison de l'analyse. Il faut donc créer de nouvelles règles pour les transformer en récursivités droites.

$$\begin{aligned} E &\rightarrow F E_{\text{aux}} \\ E_{\text{aux}} &\rightarrow + F E_{\text{aux}} \mid - F E_{\text{aux}} \mid \epsilon \\ F &\rightarrow P F_{\text{aux}} \\ F_{\text{aux}} &\rightarrow * A F_{\text{aux}} \mid / A F_{\text{aux}} \mid \epsilon \\ A &\rightarrow \text{constante} \mid ( E ) \end{aligned}$$

## 2.4 Ajout de l'opérateur de puissance

Ensuite, pour ajouter un opérateur de puissance, de telle sorte qu'il soit prioritaire sur la multiplication (mais pas sur les parenthèses) et associatif à droite, on ajoute une fonction supplémentaire entre les facteurs et les atomes. Comme elle utilise la récursivité droite, il n'est pas nécessaire de la transformer comme les précédentes. On obtient donc la grammaire :

$$\begin{aligned} E &\rightarrow F E_{\text{aux}} \\ E_{\text{aux}} &\rightarrow + F E_{\text{aux}} \mid - F E_{\text{aux}} \mid \epsilon \\ F &\rightarrow P F_{\text{aux}} \\ F_{\text{aux}} &\rightarrow * P F_{\text{aux}} \mid / P F_{\text{aux}} \mid \epsilon \\ P &\rightarrow A ^ P \mid A \\ A &\rightarrow \text{constante} \mid ( E ) \end{aligned}$$

## 2.5 Changement de la règle de puissance

Cependant, la documentation des **stream parsers** précise que pour une même fonction, on ne peut pas commencer deux schémas par les mêmes mot-clés. Or,  $P$  se décompose en  $A \wedge P$  ou  $A$ , qui commencent tous deux par  $A$ . On peut utiliser les paramètres d'exception **Stream.error** pour se rendre compte qu'en effet, l'analyse descendante attendra le reste de la règle ( $\wedge P$ ) même si l'expression devrait utiliser le deuxième schéma de  $P$ .

Pour résoudre ce problème, on décompose la règle en deux règles :

$$\begin{aligned} P &\rightarrow A P\_aux \\ P\_aux &\rightarrow \wedge P \mid \epsilon \end{aligned}$$

## 2.6 Grammaire finale

Ainsi, la grammaire obtenue est :

$$\begin{aligned} E &\rightarrow F E\_aux \\ E\_aux &\rightarrow + F E\_aux \mid - F E\_aux \mid \epsilon \\ F &\rightarrow P F\_aux \\ F\_aux &\rightarrow * P F\_aux \mid / P F\_aux \mid \epsilon \\ P &\rightarrow A P\_aux \\ P\_aux &\rightarrow \wedge P \mid \epsilon \\ A &\rightarrow constante \mid ( E ) \end{aligned}$$

## 3 Implémentation

La grammaire obtenue n'a plus qu'à être recopiée avec la syntaxe des **stream parsers**.

On notera tout de même qu'il convient d'envoyer en argument l'expression déjà lue par une fonction principale aux fonctions auxiliaires, qui en auront besoin pour construire l'expression.

Par exemple, pour l'expression "1+2", on utilise la première règle :  $E \rightarrow F E\_aux$  dans laquelle  $F$  se réduira en 1 et  $E\_aux$  en + 2. La fonction  $E\_aux$  doit donc prendre en argument le facteur reconnu par la règle  $F$  (ici, 1) pour pouvoir reconstituer l'arbre de l'expression plus(1,2) après avoir reconnu + 2.

## 4 Vérification

Une fois le parseur implémenté, nous avons souhaité le tester pour vérifier sa correction. Dans un premier temps, nous avons vérifié chaque associativité, chaque priorité en donnant divers exemples. Mais nous avons souhaité automatiser le processus de vérification.

Ainsi, nous avons donc un évaluateur, qui à partir d'une expression (sous forme d'arbre syntaxique abstrait), calcule le résultat.

Dans un second temps, nous avons créé un générateur aléatoire d'expressions arithmétiques (sous forme de chaînes de caractères) en **Ocaml**. On ne génère ainsi que des expressions arithmétiques correctes (pas d'erreur de parenthésage ou de caractères non reconnus).

Enfin, un programme en **bash** appelle le générateur et récupère une chaîne de caractères. Puis cette chaîne de caractère est scannée, analysée puis évaluée par notre parseur. D'un autre côté, on

évalue également l'expression en **Python** (qui utilise les mêmes priorités et associativité que celles de notre parseur). Enfin, on compare les résultats.

## 5 Résultats

Il a ainsi été possible d'effectuer des tests en grande quantité (plusieurs centaines). Les erreurs proviennent de divisions par zéro et de dépassements mémoires dans notre évaluateur en **Ocaml** (ces dernières ne sont pas signalées par l'évaluateur mais donnent un résultat incohérent). D'autres erreurs proviennent quand on effectue une division avec des nombres négatifs, puisque **Python** et **Ocaml** n'arrondissent pas dans le même sens. Mais cette erreur provient d'une différence d'évaluateur, non de parseur.

La vérification automatique nous a également permis de trouver une erreur dans le fichier original utilisant **Genlex** pour scanner la chaîne originale. Par exemple, pour la chaîne "(1-0)-1", on obtient une erreur lors de l'analyse descendante. En utilisant les paramètres d'exception **Stream.error**, on constate que le parseur fait une erreur alors qu'il attend une parenthèse fermante, tandis que le parenthésage de l'expression est correct. Utiliser l'autre **lexer** résout le problème. Nous n'avons pas réussi à expliquer ce problème, seulement à identifier qu'elle apparaît chaque fois que la chaîne de caractère contient la sous-chaîne "0)-" .

Il ne s'agit pas d'une vérification exhaustive de toutes les expressions correctes que pourraient analyser notre parseur. En effet, nous avons ajouté des conditions de profondeur maximale dans la génération d'une expression, pour éviter les dépassements mémoires la plupart du temps. Cependant le nombre élevé de tests effectué n'a pas pu mettre en défaut notre parseur.

## 6 Conclusion

Pour réaliser ce parseur, il nous a fallu nous intéresser au fonctionnement de l'analyse descendante, pour adapter nos règles à la méthode utilisée (pas de récursivités gauches par exemple). Il a fallu également s'adapter à l'outil utilisé, les **stream parsers** de **Camlp4** (pas de schémas qui commencent par la même séquence).

Le développement d'un outil de vérification empirique nous a permis de nous rendre compte de certains problèmes dans notre implémentation, et de nous convaincre de sa robustesse.