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 par 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 multplication 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

On souhaite d'abord 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.

```
E -> E + F | E - F | F
F -> F * A | A
A -> constante | ( E )
```

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 -> F / A`.

```
E -> E + F | E - F | F
F -> F * A | F / A | A
A -> constante | ( E )
```

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.

```
E      -> F E_aux
E_aux  -> + F E_aux | - F E_aux | ϵ
F      -> A F_aux
F_aux  -> * A F_aux | / A F_aux | ϵ
A      -> constante | ( E )
```

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 :

```
E      -> F E_aux
E_aux  -> + F E_aux | - F E_aux | ϵ
F      -> P F_aux
F_aux  -> * P F_aux | / P F_aux | ϵ
P      -> A ^ P | A
A      -> constante | ( E )
```

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 ^ 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 (^ P) même si l'expression devrait utiliser le deuxième schéma de P.

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

$$\begin{array}{ll} \text{P} & \text{--> A P\_aux} \\ \text{P\_aux} & \text{--> ^ P | } \epsilon \end{array}$$

De fait, nous avons du transformé notre grammaire en une grammaire LL(1) pour pouvoir respecter la spécification des `stream parsers`.

2.6 Grammaire finale

Ainsi, la grammaire obtenue est :

$$\begin{array}{ll} \text{E} & \text{--> F E\_aux} \\ \text{E\_aux} & \text{--> + F E\_aux | - F E\_aux | } \epsilon \\ \text{F} & \text{--> P F\_aux} \\ \text{F\_aux} & \text{--> * P F\_aux | / P F\_aux | } \epsilon \\ \text{P} & \text{--> A P\_aux} \\ \text{P\_aux} & \text{--> ^ P | } \epsilon \\ \text{A} & \text{--> constante | (E)} \end{array}$$

3 Implémentation

La grammaire obtenue n'a plus qu'à être recopiée avec la syntaxe des `stream parsers` qui permet de modéliser assez naturellement les différentes productions d'une grammaire. On doit cependant ajouter un nouveau mot-clé au lexer pour chaque opération que l'on désire implémenter.

On notera par ailleurs 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 --> 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.

En réalité, les `stream parsers` nous permettent de manipuler une grammaire attribuée. L'attribut d'un noeud de l'arbre d'analyse syntaxique est dépendant de celui du fils droit, lui-même dépendant de celui du fils gauche, c'est pourquoi on doit fournir l'attribut synthétisé par le fils gauche au parser pour qu'il puisse calculer l'attribut du fils droit, puis celui du noeud étudié.

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 adviennent 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'il 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 pourrait analyser notre parseur. En effet, nous avons ajouté des conditions de profondeur maximale dans la génération d'une expression, pour éviter les dépassements mémoire la plupart du temps. Cependant le nombre élevé de tests effectués 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, `Camlp4`, qui permet de modéliser assez facilement une grammaire en représentant les productions par des `stream parsers`, et même de manipuler des grammaires attribuées en utilisant les arguments renvoyés par les `parsers` comme des attributs. Cependant cet outil présentent certaines limites, comme l'impossibilité d'utiliser des schémas commençant par la même séquence dans un `parser`, ce qui contraint à utiliser des grammaires LL(1). On notera aussi que la que la lecture par la gauche propre aux `streams` est adaptée à l'utilisation de l'analyse descendante, ce qui nous convenait ici, mais pas à celle de l'analyse ascendante.

Enfin, 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.