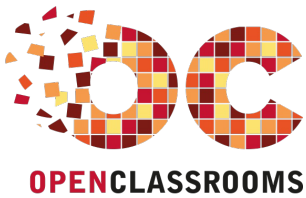
[OCaml] Les flots de données

Par Olivier Strebler (shareman)



www.openclassrooms.com

Dernière mise à jour le 10/04/2012

Sommaire

[OCaml] Les flots de données

Par  Olivier Strebler (shareman)

Mise à jour : 10/04/2012
Difficulté : Intermédiaire

Les flots de données (les *stream* d'OCaml) sont des outils très puissants et assez particuliers permettant, entre-autres, d'implémenter facilement des analyseurs syntaxiques descendants. Dans ce tutoriel, nous allons découvrir de quoi il retourne exactement avec une première sous-partie consacrée aux explications théoriques, une deuxième touchant plus le côté pratique et une troisième, plus dense, où l'on étudiera un exemple concret de programme réalisé à l'aide des *stream*.
Sommaire du tutoriel :

Présentation des stream

Présentation générale

Le flot de données, dit stream dans le langage fonctionnel OCaml, est une structure de données d'un genre peut-être assez nouveau pour certains : c'est une structure abstraite. En gros, cela signifie que l'on ne connait pas son implémentation. Ceci est signalé par OCaml à l'aide de "<abstr>" que l'on reverra plus tard. Il peut y avoir plusieurs raisons à garder une implémentation "abstraite" :

- Soit le type n'est pas représentable avec les moyens mis à disposition par OCaml pour le construire ;
- Soit l'implémentation est volontairement cachée pour limiter le programmeur aux fonctions de manipulation du type déjà toutes faites.

Comme nous l'avons vu, les stream sont ce que l'on appelle des flots de données. Concrètement, un flot de données est en réalité une suite d'éléments du même type. Attention à ne pas confondre stream, list (liste chaînée) et array (tableau). Il peut être impossible d'arriver à la fin de cette suite, elle peut donc être infinie. En effet, les stream ont quelque chose de particulier : leur caractère paresseux. Cela signifie que chaque élément d'un stream n'est calculé que lorsqu'on le demande. Cette technique possède ses avantages et ses défauts et l'un des grands avantages, que n'ont pas les listes ou les tableaux par exemple, c'est qu'il est possible de créer des stream infinis. Autre caractère notable chez les stream : leur caractère destructif. Ici, cela veut simplement dire qu'une fois qu'un élément d'un stream a été reconnu, il est automatiquement supprimé du flot de données. Cela possède ses avantages et ses défauts aussi. Le regroupement du caractère paresseux et du caractère destructif dans le type stream a une importante conséquence :

- Quand un élément vient d'être calculé, c'est qu'on a cherché à le reconnaitre et si on l'a reconnu, la définition veut qu'il soit immédiatement retiré du stream. Finalement, on constate que le stream en soi ne contient jamais rien, il retient juste comment il doit calculer le prochain élément ;

Stream et analyse syntaxique descendante

Les stream sont particulièrement utilisés dans les implémentations en OCaml d'analyseurs syntaxiques descendants. En effet, leurs caractéristiques et les fonctionnalités des extensions de camlp4 qui vont avec en font un outil de premier choix. En réalité,

certains développeurs de compilateurs n'utilisent OCaml que pour cette structure de données. Nous allons d'ailleurs étudier un "bout" de mini-compilateur dans la troisième et dernière sous-partie.

OCaml met en réalité deux outils particulièrement puissants à disposition pour la construction d'analyseurs syntaxiques :

- ocamlyacc qui n'est autre qu'une implémentation OCaml du célèbre générateur d'analyseurs syntaxiques Yacc (ou Bison, l'équivalent libre de Yacc). ocamlyacc produit un analyseur syntaxique ascendant, qui part des unités lexicales du programme pour retrouver l'axiome de la grammaire décrivant le langage. Pour plus d'informations, les bons sites ne manquent pas ;
- Bien entendu, les stream. Comme nous l'avons déjà évoqué, on utilise les stream pour les analyses syntaxiques descendantes, plus simples à implémenter mais généralement moins efficaces que les analyses syntaxiques ascendantes. Dans une telle analyse, on part de l'axiome de la grammaire décrivant le langage pour arriver aux unités lexicales du programme.

Naturellement, ce qui vous intéresse, c'est de savoir comment s'y prendre pour créer et manipuler les stream. C'est au programme de la deuxième sous-partie.

Manipuler les streams

Tout au long du cours, je vais m'appuyer sur l'interpréteur interactif d'OCaml, c'est-à-dire le programme ocaml. Pour compiler avec ocamlc (qui produit du bytecode qui peut être interprété avec ocamlrun), la commande est "ocamlc -pp "camlp4o" source.ml". Pour le compilateur natif ocamlopt, c'est exactement la même chose.

Stream et camlp4

Quand on parle de stream en OCaml, on pense souvent à l'extension du langage camlp4. Mais en réalité, le type abstrait du stream est déjà présent dans le noyau stable d'OCaml et est manipulable grâce aux fonctions fournies par le module Stream. Ce n'est cependant pas très pratique et c'est très limité. Voici un exemple de code créant un stream vide sans utiliser l'extension camlp4 :

Code : OCaml

```
# let stm = Stream.of_list [] ;;
val stm : '_a Stream.t = <abstr>
```

Comme on peut le voir, le type du stream en OCaml est en réalité le type Stream.t. Remarquez la signalisation de l'abstraction par <abstr>. Manipuler les stream uniquement avec le module Stream est une tâche très lourde et lassante (nous allons cependant revenir sur ce module à la fin de cette sous-partie). C'est l'une des raisons pour lesquelles les développeurs d'OCaml mettent à disposition l'extension camlp4. "camlp4" signifie "Caml Preprocessor and Pretty-Printer". camlp4 fournit entre-autres une extension de la syntaxe d'OCaml pour faciliter la création et la manipulation des stream, extension qui va nous intéresser ici et dont on va se servir pour le reste du cours.

Ce qui vous intéresse maintenant, c'est de savoir comment utiliser l'extension de la syntaxe pour les stream. Pour la dernière version d'OCaml, il va vous falloir charger deux choses : dynlink.cma puis camlp4o.cma, dans cet ordre précisément.

Code : OCaml

```
# #load "dynlink.cma" ;;
# #load "camlp4o.cma" ;;
```

Si tout se passe comme prévu, vous devriez vous retrouver avec un message "Camlp4 Parsing version 3.11.0" (avec votre version d'OCaml si elle est différente de la mienne). Maintenant que cela est fait, camlp4 vous met à disposition une syntaxe de manipulation des stream très proche des array et des list. Pour rappel, la syntaxe pour les tableaux est "[| |]" et pour les listes, il s'agit de "[ ]", et pour les deux (les trois en fait, pour les stream il en sera de même comme nous allons le voir), on sépare chaque élément avec un point-virgule ';'. La syntaxe pour les stream est "[< >]" :

Code : OCaml

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

Le flot de données vide s'écrit "[< >]". La manipulation des stream en utilisant cette syntaxe très légère est cependant assez différente de celle des array ou des list sur deux principaux points :

- Chaque élément d'un stream doit être précédé d'une apostrophe ', justement pour signaler qu'il s'agit bien d'un élément ;
- On introduit la notion de sous-flot. Quand on décrit un stream avec la syntaxe "[< >]", le sous-flot est signalé par l'absence de l'apostrophe.

En gros, en écrivant "[< 'a >]", a est un élément du flot, mais en écrivant [< a >], a est considéré comme un sous-flot. Les sous-flots ne sont en aucun cas à considérer comme des "entités" d'un flot plus grand, ils ne représentent que des parties du flot principal (le terme "sous-flot" peut donc porter à confusion). Le cas du flot vide "[< >]" est particulier. Dans n'importe quel stream, on peut considérer [< >] comme un sous-flot omniprésent. Ainsi, "[< [<>]; 'a; [<>] >]" est *strictement* équivalant à "[< 'a >]". Petit essai :

Code : OCaml

```
# let stm = [< '4 >] ;;
val stm : int Stream.t = <abstr>
# let stm2 = [< stm; '5 >] ;;
val stm2 : int Stream.t = <abstr>
# let stm3 = [< stm; 'true >] ;;
Characters 19-24:
  let stm3 = [< stm; 'true >] ;;
                     ^^^^^
Error: This expression has type bool Stream.t but is here used with
type
         int Stream.t
```

Au terme de ce code, stm est un stream d'entier contenant 4 et stm2 est un stream d'entier contenant 4 et 5, dans cet ordre. Remarquez qu'il est donc extrêmement simple de profiter du concept du sous-flot dans la concaténation de flot. Dans le précédent code, "[< stm; '5 >]" est donc strictement égal à "[< '4; '5 >]". Dans cet exemple, on retrouve aussi le typage fort d'OCaml : impossible de concaténer un stream d'int et un stream de bool et ce type d'erreur est signalé bien avant l'exécution du code.

Il faut également souligner une autre particularité dans la manipulation des stream avec camlp4, qui se trouve dans le filtrage de motif (pattern matching). Il faudra être plus vigilant que pour le filtrage de listes par exemple. Premièrement, il faut savoir que camlp4 introduit un nouveau mot-clef pour le filtrage de motif de stream : "parser". En effet, on écrit plus "match stm with" suivi des différents cas possibles mais "match stm with parser". Le raccourci bien connu du "function" est géré en utilisant "parser" tout seul et en omettant le dernier paramètre (le principe est en fait le même que "function" : on génère une fonction à un paramètre et on le "match" directement ; dans le cas de "parser", ce paramètre est forcément un stream). Deuxièmement, il faut savoir que chaque élément reconnu est retiré du flot, même si le reste du *match case* concerné peut potentiellement ne plus coller. Voici un petit exemple de filtrage :

Code : OCaml

```
let test stm = match stm with parser
| [< '1; '2 >] -> "un deux"
| [< >] -> "autre"
```

Ce qui est strictement la même chose que :

Code : OCaml

```
let test = parser
| [< '1; '2 >] -> "un deux"
```

```
| [< >] -> "autre"
```

Il faut également savoir que dans un tel pattern matching, un flot peut être reconnu par un flot plus petit si ce dernier correspond à une partie gauche du flot d'entrée. Par exemple, le flot "[< '1; '2; '3 >]" sera reconnu par le *match case* "[< '1; '2 >]" si ce dernier est évalué.

Encore une fois, le flot de données vide "[< >]" est à utiliser avec des pincettes. En effet, il ne faut pas le confondre avec la liste vide "[]" par exemple car il filtre *tous* les stream. En effet, n'importe quel stream "[< s >]" est équivalent à "[< [< >]; s >]", et par conséquent "[< >]" est une partie gauche de tous les stream. Il représente en réalité le cas "autre" ("_" n'est pas adapté pour les stream). Il doit ainsi se retrouver en dernier *match case* du filtrage. En l'omettant, vous n'aurez cependant aucun warning à la compilation comme on en retrouve dans d'autres cas analogues.

Gérer les erreurs

Bien gérer les erreurs, c'est primordial. Que se passe-t-il si aucun des match case du filtrage n'a été reconnu, et n'a ni même entamé en reconnaissance ? Ce genre de cas est géré à l'aide des exceptions et il y a en a précisément deux qui vont nous intéresser : Stream.Failure et Stream.Error. Stream.Failure est lancée quand aucun match case n'est un possible candidat pour le stream filtré, c'est-à-dire quand aucun premier élément du filtrage ne lui correspond. Stream.Error est beaucoup plus subtil : une telle exception est lancée quand un match case a partiellement reconnu l'entrée, mais que la suite ne colle plus. Cette dernière exception est intéressante dans les cas où l'on doit signaler précisément au programmeur ce qui ne va pas dans un stream donné. En effet, elle possède un attribut de type string, qui si rien n'est précisé est la chaîne vide "". En revanche, là où ça devient bénéfique, c'est quand on précise soi-même un message d'erreur. C'est possible avec l'opérateur "??" et ça se fond totalement dans la syntaxe du stream.

La syntaxe est simple : on sépare l'élément du message d'erreur associé à l'aide de "??". Exemple : la fonction suivante prend en entrée un stream, le filtre avec le motif [< '1; '2 >] est nous indique qu'il manque "2" si "1" a effectivement été reconnu.

Code : OCaml

```
let test = parser
| [< '1; '2 ?? "Il manque 2 !" >] -> "un deux"
| [< >] -> "autre"
```

Algorithmes sur les stream

Le pattern matching de stream va encore beaucoup plus loin et vous allez encore découvrir certaines nouveautés dans cette section. Cette dernière est consacrée à la présentation d'algorithmes couramment utilisés lors de la manipulation de flot de données. Certains algorithmes seront déjà présents dans le module Stream mais il faut bien comprendre que le but ici est purement pédagogique, le but final étant de vous rendre à l'aise avec les stream. De plus, vous aurez ici un petit avant-goût de la puissance des stream.

Nous allons commencer par l'exemple classique : récupérer et retirer le premier élément d'un flot. Comme vous vous en doutez, ces deux actions n'en sont en réalité qu'une. Le nom traditionnel pour cette fonction est "next". Nous allons procéder simplement en considérant un seul cas possible qui sera "[< 'e >]" pour e représentant donc l'élément de tête. Le seul cas où ça ne passera pas est le flot vide, auquel cas une exception est lancée. On obtient donc le simple code suivant :

Code : OCaml

```
let next = parser [< 'e >] -> e
```

Un autre exemple classique est l'implémentation d'une fonction "map" adapté aux stream. Pour rappel, le principe de "map" (que l'on retrouve aussi dans le module List par exemple) est d'appliquer une fonction à chaque élément du conteneur d'entrée et de renvoyer un conteneur semblable mais contenant le retour des différents appels. Dans notre cas, nous allons donc utiliser une fonction récursive qui prend en argument le flot d'entrée et une fonction et qui construira le flot de sortie attendu. La définition fonctionnelle de cette fonction sera donc la suivante : "si le flot est un élément suivi d'un reste, le résultat *est* le flot contenant le retour de la fonction qui prend l'élément en paramètre suivi du (sous-) flot de sortie de l'appel récursif à "map" sur le reste; si le

flot est vide, le résultat *est* le flot vide". L'idée peut donc très simplement se traduire en OCaml :

Code : OCaml

```
let rec map_stream f = parser
| [< 'e; r >] -> [< 'f e; map_stream f r >]
| [< >] -> [< >]
```

À présent, nous allons nous intéresser à une autre fonction qui prend en paramètre un stream et qui renvoie son inverse, c'est à dire, en termes plus simples, la même suite d'éléments mais dans l'autre sens. En utilisant le concept du sous-flot, l'idée de l'algorithme est très simple : "l'inverse d'un flot contenant un élément suivi un reste *est* le flot contenant l'inverse du reste suivi de l'élément; l'inverse du flot vide est le flot vide". En OCaml, on a donc :

Code : OCaml

```
let rec reverse = parser
| [< 'e; r >] -> [< reverse r; 'e >]
| [< >] -> [< >]
```

Tout à l'heure, j'ai parlé de nouveautés. Effectivement, un concept très puissant peut-être utilisé pour réécrire la précédente fonction reverse. En effet, on peut appeler une fonction et récupérer son retour au sein même d'un *match case* (retour qui n'est pas forcément de type Stream.t mais dans notre cas si). En résumé, le plus gros du travail peut-être effectué avant "->". Je crois qu'un exemple est plus compréhensible :

Code : OCaml

```
let rec reverse = parser
| [< 'e; a = reverse >] -> [< a; 'e >]
| [< >] -> [< >]
```

La variable a contient le retour de la fonction reverse qui prend en paramètre le fameux reste qu'on a laissé implicite. En gros, ici on met aussi l'accent sur la définition suivante : "l'inverse d'un stream non vide est l'élément de tête précédé de l'inverse du reste", sauf qu'on définit cet inverse avant l'expression de retour. Nous réutiliserons cette technique assez pratique dans la troisième sous-partie.

Enfin, pour finir cette section, intéressons-nous à un dernier algorithme quasiment aussi simple à implémenter que next : l'algorithme de concaténation. Comme vous vous en doutez, le résultat de la concaténation d'un stream stm1 avec un stream stm2 peut être considéré comme un stream dont stm1 et stm2 sont deux sous-flots qui se succèdent. Encore une fois, en OCaml, c'est très simple :

Code : OCaml

```
let concat stm1 stm2 = [< stm1; stm2 >]
```

Module Stream

Le module Stream contient plusieurs fonctions assez pratiques et toujours bon à savoir utiliser. L'utilité des fonctions de la bibliothèque standard est justement de ne pas être ré-implémentées dans chaque code, il faut les utiliser ! Je ne vais pas détailler le nom, le type et la raison d'être de chaque fonction de ce module ici, un tutoriel n'est pas une doc, c'est à vous de vous renseigner ici en cas de besoin.

Parsons du préfixé !

Avant-propos

L'objectif de cette dernière sous-partie est de vous donner un petit exemple concret de cas où l'on peut être amené à utiliser les stream pour l'analyse syntaxique par descente récursive (ou analyse descendante). Elle fait donc office d'illustration. Ainsi, nous allons réaliser ici un petit début de compilateur. Le thème n'est pas la compilation, sachez simplement qu'une partie frontale de compilateur comporte au moins les phases suivantes : analyse lexicale, analyse syntaxique, analyse sémantique et production de code intermédiaire. Dans le cas présent, nous retrouverons l'analyse lexicale, que nous allons voir très rapidement et bien sûr l'analyse syntaxique qui est au centre de la sous-partie. Au lieu de produire du code, nous allons évaluer l'arbre produit par l'analyse syntaxique.

Notre objectif est simple : nous allons chercher à élaborer un programme en trois étapes qui devra prendre en entrée une string contenant un calcul suivant la notation d'expressions mathématiques préfixées. Pour en savoir plus : ici. Retenez que dans cette notation, les opérateurs se trouvent avant leurs opérandes. Ainsi, "* + 4 5 6" est équivalent à "(4 + 5) * 6" en notation infixée. Cette notation a l'avantage de pouvoir être plus facilement analysée avec une analyse syntaxique descendante. En effet, on ne se heurte pas au problème de la récursivité à gauche que l'on retrouve avec la notation infixée et postfixée et on peut donc pleinement se pencher sur les stream.

Les trois étapes mentionnées précédemment seront les suivantes :

- L'analyse lexicale : on découpe la chaîne d'entrée en unités lexicales, c'est-à-dire en entités lexicales qui ont un sens dans le langage. Le "langage" (dans la mesure où l'on peut définir son lexique, sa syntaxe et son sens) qui nous intéresse ici est la notation mathématique préfixée. On doit donc identifier les nombres et les opérateurs ;
- L'analyse syntaxique : on vérifie que la syntaxe du langage est respectée dans le flot d'unités lexicales provenant de l'analyse lexicale. Cette analyse retourne un arbre syntaxique abstrait symbolisant la structure syntaxique du "programme" avec les éléments utiles pour la suite. On utilisera un cas particulier d'analyse descendante : une analyse syntaxique par descente récursive et on va se servir pour cela des stream ;
- Le calcul du résultat : dans notre cas, ce qui nous intéresse est d'obtenir le résultat du calcul. Nous allons faire cela récursivement à partir de l'arbre abstrait renvoyé par l'analyse syntaxique.

L'analyse lexicale et le calcul du résultat ne sont pas au centre de cette section, nous mettons l'accent sur l'analyse syntaxique comme le but est de montrer l'intérêt des stream dans une telle situation. Nous allons donc rapidement coder un petit analyseur lexical. Sans se prendre la tête, nous allons utiliser le module Genlex et plus précisément la fonction make_lexer qui prend en paramètre une string list contenant les mots-clef et un char Stream.t contenant le flot de caractère entrant. Pour nous simplifier la tâche, pour ce code et pour les codes suivants, nous allons utiliser une instruction "open" pour Genlex. On obtient :

Code : OCaml

```
open Genlex

let lexer str =
  let kwd = ["+"; "-"; "*"; "/"; "^"]
  in make_lexer kwd (Stream.of_string str)
```

Cette fonction renvoie la suite d'unités lexicales correspondante à la chaîne d'entrée, qui seront de type Genlex.token.

Stream : l'analyse syntaxique

La notation préfixée des expressions est un langage. Comme tout langage, on peut définir sa syntaxe de manière pertinente et non-contextuelle à l'aide des BNF, ou grammaires non-contextuelles. Nous allons définir des règles qui décrivent comment un programme, à partir d'une règle principale appelée axiome, peut être structuré. Pour les expressions préfixées, il faut se poser la question : quelle forme peut prendre une expression ? "6", "+ 34.15 3", "* - 4 2.1 27". On remarque qu'une expression est soit un nombre (int ou float), soit un opérateur suivi de deux expressions. Très simplement, on peut décrire cela à l'aide de deux règles :

Citation

expr -> **op** *expr expr* | *val*

val -> **int** | **float**

"*expr*" et "*val*" sont des règles syntaxiques et le tout forme une (très) petite grammaire dont "*expr*" est l'axiome.

La barre verticale exprime un "ou". "*expr*" est donc une variable pour désigner les expressions d'une manière générale, tandis que "val" définit la forme que peut prendre une unique valeur. Dans la grammaire ci-dessus, j'utilise une norme assez répandue : les non-terminaux sont écrits en italique et les terminaux en gras. Dans notre cas, il y a trois types de terminaux : les nombres entiers, les nombres flottants et les opérateurs, ce qui est clairement visible. Maintenant pour en venir au principal, les stream, il faut savoir qu'à l'aide de ces derniers, l'implémentation de l'analyseur syntaxique reflètera de manière extrêmement transparente la grammaire que nous venons de voir. Les stream sont conçus pour écrire de manière intuitive un analyseur syntaxique à partir de BNF. On trouve donc un lien fort en théorie (les BNF) et pratique (les stream) qui est tout à notre avantage.

Il reste un dernier point à éclaircir avant de nous ruer sur l'implémentation : les arbres syntaxiques abstraits. Ces arbres (AST) sont ceux produits par l'analyseur syntaxique. Ils représentent la structure syntaxique du programme, dans lequel chaque opérateur (le terme doit être pris au sens large ici) est un nœud et chaque opérande l'un de ses fils. Les feuilles représentent donc les terminaux. Dans un tel arbre, on ne garde que les parties "nécessaires" pour la suite. En gros, quand on construit un AST à partir d'une production "*instr* -> **if (** *expr* **)** *instr* **else** *instr*" par exemple, les parties intéressantes du nœud créé seront expr et les deux instr. On peut laisser le if, les parenthèses et le else de côté, on peut les deviner à l'aide du nom du nœud et de toute façon, ils ne nous intéresseront plus.

Dans notre exemple, nous voulez en outre mélanger "float" et "int" au sein d'une même expression sans conversion explicite. On peut facilement convertir un "int" en "float" sans perdre d'information, du coup ce qui se passe après l'analyse syntaxique traitant uniquement des "float", cette dernière prendra soin de convertir les "int" s'il y en a.
L'arbre abstrait que nous allons devoir construire devra donc gérer deux cas : le nœud à construire est un "float" *ou* le noeud à construire est un opérateur arithmétique et deux AST. Nous allons directement utiliser le type des opérateurs arithmétiques sur "float" d'OCaml, à savoir (float -> float -> float), pour représenter l'opérateur et non une string, pour ne plus avoir à s'en préoccuper lors du calcul du résultat. En OCaml, on a donc simplement :

Code : OCaml

```
type tree =
| Lf of float
| Nd of (float -> float -> float) * tree * tree
```

Les opérateurs que nous avons définis en mots-clef seront renvoyés par l'analyse lexicale sous le "sous-type" Kwd of string. Au mieux, on récupère donc une string lors de l'analyse syntaxique. Pour obtenir le bon opérateur en fonction de cette string, nous allons juste construire une petite fonction toute simple :

Code : OCaml

```
let to_op op = match op with
| "+" -> ( +. ) | "-" -> ( -. )
| "*" -> ( *. ) | "/" -> ( /. )
| "^" -> ( ** )
| _ -> failwith (op ^ " : non connu")
```

À présent, nous pouvons nous pencher sur l'implémentation de l'analyseur syntaxique. Je rappelle avant que nous aurons affaire à une analyse syntaxique par descente récursive. Ici, on n'aura pas de problème car notre grammaire n'est pas récursive à gauche et en plus de cela, elle est prédictive LL(1) (ce qui signifie qu'il nous suffit d'observer le premier élément pour choisir la bonne production).

Au niveau de la conception, c'est très simple : on crée typiquement une fonction par règle syntaxique, chacune des ces fonctions prenant un stream en paramètre et renvoyant un AST. Nous allons ainsi créer deux fonctions : "expr" et "val_". On applique ensuite un pattern matching sur le stream d'entrée. Si l'on retrouve la production "*expr* -> **op** *expr expr*", on renvoie un Nd (pour "node") contenant l'opérateur OCaml associé à l'opérateur représenté par une string suivi des deux AST correspondant aux retours des deux appels récursifs à "expr" nécessaires (on va bien sûr faire cela directement dans le match case). Si l'on retrouve la production "*val* -> **int**", on renvoie simplement un Lf (pour "leaf") stockant l'entier qu'on aura pris soin de convertir en "float". Si l'on retrouve "*val* -> **float**", on fait la même chose, mais sans convertir.

Les opérateurs stockés dans le stream seront de sous-type Kwd (voir module Genlex), les "float" de sous-type Float et les "int" de sous-type Int. Nous n'allons pas implémenter le cas "autre", qui serait bien sûr une erreur de syntaxe ici. Une exception Stream.Failure sera lancée si au aucun match case n'est reconnu. Au terme de cette analyse, un AST complet est renvoyé. Voici

l'implémentation, qui suit la précédente description :

Code : OCaml

```
let rec expr = parser
| [< 'Kwd k; e1 = expr; e2 = expr >] -> Nd (to_op k, e1, e2)
| [< v = val_ >] -> v

    and val_ = parser
| [< 'Int i >] -> Lf (float_of_int i)
| [< 'Float f >] -> Lf f
```

Vous remarquez donc que c'est vraiment très concis. Il ne fait aucun doute qu'il est évidemment impossible d'être aussi concis et clair à la fois en C++ ou en Java par exemple. Comme implémentation, on ne peut plus simple, c'est ce qui fait toute la puissance d'OCaml et des stream pour l'analyse syntaxique.
Un dernier point : il faut prendre garde à une chose un peu subtile. Je vais prendre un exemple : et si j'écris "45. +", ce qui est visiblement syntaxiquement faux, l'analyse lexicale nous renverra le stream "[< 'Float 45.; 'Kwd "+" >]". Ce dernier sera reconnu par "val_", puis "expr". "expr" renverra un AST contenant juste la valeur 45, l'analyse syntaxique est terminée et le stream n'est pas vide et... aucune erreur ! Il faut y remédier et l'idée classique est de vérifier après l'analyse syntaxique s'il reste des éléments dans le stream. À vous de jouer ;).

Et on teste

On peut facilement construire une fonction qui par récursions calcule le résultat à partir de l'AST provenant de l'analyse syntaxique. Il nous suffit de choisir la bonne action en fonction du sous-type. Si nous avons affaire à Lf, on renvoie le "float" stocké lui-même. Si nous avons Nd, on applique l'opérateur stocké entre le retour de result sur le premier sous-arbre et le retour de result sur le second sous-arbre. Au final, on obtient le résultat du calcul de départ.

Code : OCaml

```
let rec result = function
| Lf f -> f
| Nd (op, e1, e2) ->
  op (result e1) (result e2)
```

Voici un petit exemple qui montre que notre objectif a été atteint :

Code : OCaml

```
# result (expr (lexer "* + 22.1 5 - 10 3.2")) ;;
- : float = 184.28
```

Mieux gérer les erreurs

D'une manière générale en compilation, la gestion des erreurs est fondamentale. En cas d'erreur, on doit pouvoir obtenir un maximum d'informations sur cette dernière : la ligne (voire la (les) colonne(s)) concernée, la nature de l'erreur et pourquoi pas, des suggestions pour la corriger. Encore une fois, les streams s'y prêtent très bien : on peut le faire avec la spécification de message d'erreur à l'aide de "??" comme nous l'avons déjà vu. Par exemple : quand on a reconnu un opérateur quelconque dans "expr", mais qu'il n'y a pas d'opérande ou peut-être qu'un. On peut donc préciser "operandes de op manquants" ou "second operande de op manquant", ou "op" sera remplacé par ledit opérateur. On est donc en mesure de produire un message plus clair que Stream.Error "", qui dit non seulement ce qui est faux, mais aussi pour quel opérateur.

On peut par exemple imaginer ceci :

Code : OCaml

```
let rec expr = parser
| [< 'Kwd op
   ; e1 = expr ?? "operandes de " ^ op ^ " manquants"
   ; e2 = expr ?? "second operande de " ^ op ^ " manquant" >] -> Nd
(to_op op, e1, e2)
| [< v = val_ >] -> v

    and val_ = parser
| [< 'Int i >] -> Lf (float_of_int i)
| [< 'Float f >] -> Lf f
```

Les stream sont des outils très pratiques. Avec le petit exemple des expressions préfixées, cela ne s'est peut-être pas assez fait sentir. Essayez de construire des analyseurs plus gros, plus complets et bien sûr pour un langage source dont la syntaxe est plus complexe, car c'est bien là que les stream deviennent très intéressants. Les problèmes de la récursivité à gauche de certaines grammaires n'empêchent pas l'utilisation des stream, comme vous pouvez le constater en lisant ce tutoriel de robocop. Dans tous les cas, il y a toujours ce théorème :

"*Tout langage algébrique peut être généré par une grammaire algébrique non récursive gauche*"

Voici en bonus le stream des entiers naturels, qui a la particularité d'être **infini** :

Code : OCaml

```
let n =
  let rec n_ i = [< 'i; n_ (i+1) >]
  in n_ 0
```

Je remercie Xavinou et robocop pour leur relecture.

shareman