Chapitre 7

Flux, analyseurs lexicaux et syntaxiques

7.1 Introduction

Pour reconnaître une phrase tapée au clavier[1], il faut d'abord identifier les différents mots (ou unités lexicales, lexèmes) qui la composent. Le découpage d'une phrase en lexèmes (mots-clefs, identificateurs, nombres...) s'appelle l'analyse lexicale de la phrase.

La liste des lexèmes obtenue est ensuite fournie à un analyseur syntaxique qui va construire une représentation interne de la phase. La représentation interne généralement sous la forme d'un arbre est appelée la **syntaxe abstraite** par opposition à la **syntaxe concrète** qui est celle utilisé pour entrer au clavier la phrase à analyser.

La principale utilisation des analyseurs est la realisation de compilateurs et d'interprètes. Un compilateur traduit un langage évolué (plus ou moins :-)) (le langage source) vers un langage moins expressif (le langage cible) alors qu'un interprète évalue chaque expression à la volée.

En Caml une des façons[2] les plus simples de construire des analyseurs lexicaux (lexer) et syntaxiques (parser) est d'utiliser les flux (stream) que nous présentons maintenant en détail.

7.2 Les Flux

Comme les listes, les flux sont des conténairs d'objets mais comme nous allons le voir sur les exemples ci-dessous, leurs propriétés sont différentes.

Syntaxiquement, en Caml, un flux se caractérise par les symboles `[<` et `>]`. On peut définir un flux vide qui, comme la liste vide, est polymorphe :

```
[< >];;
- : '_a stream = <abstr>
```

Ou un flux contenant des entiers, qui est alors du type înt stream :

```
#[< '0; '1; '2 >];;
- : int stream = <abstr>
```

[1] le terme de phrase est ici très général et s'applique à toute expression d'un langage
[2] L'alternative est d'utiliser les programmes Camllex et Camlyacc.

Remarque 1.

1. Contrairement aux listes, la valeur d'un flux n'est pas affichée (en fait, comme nous le verrons, elle n'est même pas évaluée).
2. Les éléments d'un flux sont précédés du caractère `'`.

Si un élément n'est pas précédé de `"'"` il est considéré comme un sous-flux et est expansé.

Ainsi le flux suivant n'est valide que si `s` est lui-même un flux.

```
#[< s; '"def" >];;
```

comme par exemple dans

```
#let s = [< '"abc" >] in [< s; '"def" >];;
 - : string stream = <abstr>
```

Ainsi vous pouvez écrire :

```
 #[< '0; [<'1;'2>]; '3 >];;
 - : int stream = <abstr>
```

Alors que si vous essayez de faire de même avec les listes cela provoquera une erreur de typage :

```
#[0; [1; 2]; 3 ];;
Toplevel input:
>[0; [1; 2]; 3 ];;
>^^^^^^^^^^^^^^^
This expression has type int list,
but is used with type int list list.
```

Pourquoi n'est-ce pas mal typé dans le cas des flux ? Parce que les éléments entres `[<` et `>]` sont eux-même des flux. C'est pour cela que le symbole `'` est obligatoire, il sert à créer un flux avec un unique élément (il n'est valide qu'entre `[<` et `>]`). Les deux flux suivants sont équivalents.

```
 #[< '0; [<'1;'2>]; '3 >];;

 #[< '0; '1; '2; '3 >];;
```

Ainsi pour concarténer deux flux il suffit de les juxtaposer. Nous pouvons donc définir la fonction de concaténation par :

```
 #let stream_concat s t = [< s; t >];;
 stream_concat : 'a stream -> 'a stream -> 'a stream = <fun>
```

7.2.1 Création de flux

Il existe de nombreux moyens de créer des flux. Dans le cadre des analyses lexicales et syntaxiques, on s'intéressera à des flux de caractères. De tels flux peuvent être construit à la main ou obtenu en lisant les caractères sur un canal (généralement lui-même associé à un fichier par (`open_in`) C'est l'objet de la fonction suivante :

```
stream_of_channel;;
 - : in_channel -> char stream = <fun>
```

De la même manière, on peut construire le flux de caractères associé à une de chaîne de caractères par la fonction :

```
stream_of_string;;
 - : string -> char stream = <fun>

let s = stream_of_string "abc";;
s : char stream = <abstr>
```

Autre grande différence avec les listes, **les flux sont evalués paresseusement**. Les expressions qu'ils contiennent ne sont évaluées qu'à la demande, c'est-à-dire quand on chercher à y acceder. Par conséquent si une erreur se produit à l'évaluation c'est qu'elle était inévitable, en ce sens que l'expression qui a produit l'erreur devait absolument être evaluée pour pouvoir founir un résultat.

Ainsi si on essaie de construire la liste suivante, cela provoque une erreur :

```
[1;1/0];;
Uncaught exception: Division_by_zero
```

mais on peut constuire le flux contenant les mêmes élements sans provoquer d'erreur, car le second élément n'est pas évalué à la constuction. Une erreur se produira néanmoins si on cherche à accéder à cet élément.

```
[<'1;'1/0>];;
- : int stream = <abstr>
```

L'évaluation paresseuse permet de manipuler facilement des flux "interactifs" comme le flux construit par une lecture sur l'entrée standard.

Si l'évaluation n'était pas paresseuse, c'est-à-dire si, les flux étaient immédiatement complètement évalués, l'évaluation de `stream_of_channel std_in` lirait tout sur l'entrée standard, jusqu'à ce qu'une fin de fichier soit atteinte, avant de donner le contrôle au reste du programme.

7.2.2 Filtrage

Pour accéder aux éléments d'un flux on procède par filtrage mais à la différence du filtrage sur les listes, **le filtrage sur les flux est destructif**. Écrivons une fonction de filtrage qui récupère le premier élément d'un flux passé en argument et échoue si le flux est vide :

```
let next = function [< 'x >] -> x;;
next : 'a stream -> 'a = <fun>
```

et essayons-la :

```
#let s = [< '0; '1; '2 >];;
 s : int stream = <abstr>

#next s;; - : int = 0

#next s;; - : int = 1

#next s;; - : int = 2

#next s;; Uncaught exception: Parse_failure
```

Essayons-la sur un flux issu de la lecture sur un fichier :

```
let bookmark = open_in "/lockers2/prof/pons/.netscape/bookmarks.html";;
bookmark : in_channel = <abstr>
#let bookmarkFlux = stream_of_channel  bookmark;;
bookmarkFlux : char stream = <abstr>
#next bookmarkFlux;;
- : char = `<`
#next bookmarkFlux;;
- : char = `!`
#next bookmarkFlux;;
- : char = `D`
#next bookmarkFlux;;
- : char = `O`
#next bookmarkFlux;;
- : char = `C`
#next bookmarkFlux;;
- : char = `T`
#next bookmarkFlux;;
- : char = `Y`
```

En outre, l'évaluation paresseuse des flux permet la manipulation de flux potentiellement infinis. Leur éléments seront evalués à la demande. Nous pouvons par exemple construire le flux (potentiellement) infini des nombres entiers, en utilisant des effets de bord pour voir précisement quand les calculs se produisent :

```
let  nat  =
  let rec aPartirDe n =
    [< '(print_int n; print_char ` `; n);
      aPartirDe (n+1) >] in
```

```
 aPartirDe 0;;


nat : int stream = <abstr>
```

Nous notons qu'aucune impression ne s'est produite et que le programme se termine : ceci prouve qu'aucun des éléments n'a été évalué et que le flux infini n'a pas été construit. Par contre à chaque fois que l'on filtre un élément il est evalué.

```
#next nat;;
0 - : int = 0
#next nat;;
1 - : int = 1
#next nat;;
2 - : int = 2
#next nat;;
3 - : int = 3
```

Lors du premier appel à `next`, `(print_int 0; print_char ‘ ‘; 0)` est évalué, cela affiche donc `0` et renvoi la valeur `0`. Le premier élément est supprimé du flux qui vaut maintenant `[< aPartirDe 1>]`. L'appel suivant provoque l'évaluation de `(print_int 1; print_char ‘ ‘; 1)` et détruit le premier élément du flux qui vaut maintenant `[< aPartirDe 2>]`, etc.

En resumé, le motif `[ < ’x> ]` permet de filtrer le premier élément d'un flux. Après un filtrage par un tel motif, la valeur de `x` est celle du premier élement du flux et l'élément lu est physiquement supprimé du flux.

Comme dans tous les filtrages il est possible d'essayer d'identifier le terme avec plusieurs motifs, les différents cas étant alors essayés dans l'ordre. Ainsi la variante suivante de la fonction `next` filtre aussi les flux vides.

```
#let next = function
 [< ’x >] -> x

 | [< >] -> raise (Failure "empty");;

next : ’a stream -> ’a = <fun>
```

Attention même si cela peut sembler très proche du filtrage sur les listes, le motif `[<>]` a un comportement très différent du motif `[]`. Il peut en effet être unifié avec n'importe quel terme, et s'applique donc toujours. (c'est l'equivalent du motif `_` dans un filtrage sur un type somme ou un type inductif. Une conséquence de cela est que vous devez toujours le placer à la fin sinon les autres cas ne pourront jamais être appliqués.

Exercice 1. Crible d'erathostenes

Exercice 2. Nombres parfaits

Les composants des motifs de flux peuvent également être des appels de fonction (qui correspondront aux non-terminaux, dans la terminologie des grammaires). De telles fonctions sont destinées à filtrer les sous-flux du flux passés en argument : elles sont appelées sur le flux courant, et elles sont suivies d'un motif qui filtre le résultat de cet appel. Comme c'est un peu confus voyons un exemple :

On va définir un programme d'analyse syntaxique reconnaissant les suites non vides de caractères `‘a‘` de :

```
let seq_a =
   let rec seq = function
     [< ’‘a‘; seq l >] -> ‘a‘::l
    |[< >] -> []
 in function [< ’‘a‘; seq l >] -> ‘a‘::l;;

seq_a : char stream -> char list = <fun>
```

Nous avons utilisé la fonction récursive `seq` à l'intérieur du motif de flux de la première règle. Cette définition doit être lue comme :

– si le flux n'est pas vide et si son premier élément est `‘a‘`, appliquer `seq` au reste du flux. Le resultat de cet appel sera `l` et on retournera `‘a‘::l`
– autrement, échouer (soulever l'exception `Parse_failure`);

et, `seq` doit être lu de la même manière (sauf que, puisqu'il, il n'échoue jamais a cause du filtrage par `[< >]`).

De manière moins opérationelle, nous pouvons le lire comme : " une suite non vide de `‘a‘` commence par un `‘a‘`, et est suivie d'une suite éventuellement vide de `‘a‘`.

Un autre exemple est l'identification d'une suite non vide de `‘a‘` suivie d'un `‘b‘`, ou un `‘b‘`

```
#let seq_a_b = function
   [< seq_a l; ’‘b‘ >] -> l@[‘b‘]
 | [< ’‘b‘ >]-> [‘b‘];;

seq_a_b : char stream -> char list = <fun>
```

Ici, une fois qu'un `‘a‘` a été identifié, la première règle est choisie. Toute erreur dans le filtrage de la suite (lors de l'appel à `seq_a` ou lors de la reconnaissance du dernier ‘b‘) soulèvera une exception `Parse_error`, et l'analyse échouera. D'autre part, si le premier caractère n'est pas un `‘a‘`, `seq_a` soulèvera `Parse_Failure`, et la deuxième règle `([ < ’‘b‘>] -> )` sera essayée.

Ce comportement est typique des programmes d'*analyse syntaxique prédictifs*. L'analyse prédictive est une descente récursive choisissant les règles en regardant le premier token. Une fois la règles choisie, on ne peut plus revenir sur ce choix même s'il existe d'autres règles commençant par le même token.

Les liaisons issues du filtrage des motifs de flux se font séquentiellement, contrairement aux liaisons dans le filtrage des motifs classiques , qui peuvent être vu comme se produisant en parallèle. Le filtrage des flux se fait toujours de gauche à droite. Ainsi, le calcul de la somme des éléments d'un flux de nombres entiers peut être défini comme ci-dessous :

```
let rec stream_sum n = function
   [< '0; (stream_sum n) p >] -> p
 | [< 'x; (stream_sum (n+x)) p >] -> p
 | [< >] -> n;;

stream_sum : int -> int stream -> int = <fun>

#stream_sum 0 [< '0; '1; '2; '3; '4 >];;
 - : int = 10
```

La fonction `stream_sum` utilise son premier argument comme un accumulateur contenant la somme calculée jusqu'ici. L'appel (`stream_sum (n+x)`) utilise le `x` qui a été lié dans le filtrage de flux se produisant à la gauche de l'appel.

Remarque 2. Les motifs de flux ne sont légaux que dans les constructions de fonction et dans les filtrages (par `match`). Vous ne pouvez les utiliser dans un `let` comme les motifs classiques (rappelez-vous `let (x,y,z::xs)=2,3,44::(55::[]);;`).

Puisqu'un programme d'analyse syntaxique est une fonction comme n'importe quelle autre fonction, elle peut être paramétrée ou utilisée comme paramètre.

Nous allons maintenant mettre en œuvre toutes ces notions pour construire un analyseur syntaxique du langage des expressions arithmétiques.

7.3 Analyse syntaxique des expressions arithmétiques

7.3.1 Analyse lexicale

Avant d'établir un programme d'analyse syntaxique pour des expressions arithmétiques, nous avons besoin d'un analyseur lexical capable d'identifier les différents élements des phrases : les nombres, les opérations, les parenthèses *etc.* Commençons par définir un type pour les mots-clefs. Nos expressions seront construites en utilisant les quatre opérateurs de base, des parenthèses et des entiers. On introduit donc le type suivant qui defini chacune de ces catégories :

```
#type token =
   PLUS | MINUS | TIMES | DIV | LPAR | RPAR | INT of int;;
 Type token defined.
```

Puis écrivons deux petites fonctions utilitaires :

– une pour supprimer les blancs qui ne sont pas significatifs :

```
#let rec spaces = function
  [< '` `|`\t`|`\n`; spaces _ >] -> () (* \t est la tabulation *)
  | [< >] -> ();;
spaces : char stream -> unit = <fun>
```

– une qui transforme un caractère représentant un nombre (type `char`) en le nombre équivalent (type `int`). On procède en soustrayant au code ASCII du caractère le code ASCII de `0`.

```
let int_of_digit = function
 `0`..`9` as c -> (int_of_char c) - (int_of_char `0`)
 |_ -> raise (Failure "not a digit");;

int_of_digit : char -> int = <fun>
```

Remarque 3. Le mot-clef `as` permet de nommer un motif : dans le cas présen-t, la variable `c` sera liée au chiffre filtré par '0'..'9'.

Pour reconnaître des nombres entiers, nous aurons besoin d'un programme d'analyse lexical paramétré. La reconnaissance des nombres est faite par la fonction définie ci-dessous. Le parametre entier `n` représente la valeur des premiers chiffres du nombre (ceux que l'on a déjà reconnus). Lorsque l'on reconnaît le chiffre suivant on multiplie le nombre déjà reconnu par 10 et on ajoute la valeur du chiffre que l'on vient de reconnaître.

```
let rec integer n = function
     [< ' `0`..`9` as c; (integer (10*n + int_of_digit c)) r >] -> r
     |[< >] -> n;;
 integer : int -> char stream -> int = <fun>
```

Cela donne par exemple :

```
#integer 0 (stream_of_string "12345");;
 - : int = 12345
```

Nous sommes maintenant prêts à écrire l'analyseur lexical, prenant un flux de caractères en entrée, et renvoyant un flux de mots-clefs (token) en sortie.

```
#let rec lexer s = match s with
  [< '`(`; spaces _ >] -> [< 'LPAR; lexer s >]
 |[< '`)`; spaces _ >] -> [< 'RPAR; lexer s >]
 | [< '`+`; spaces _ >] -> [< 'PLUS; lexer s >]
 | [< '`-`; spaces _ >] -> [< 'MINUS; lexer s >]
 | [< '`*`; spaces _ >] -> [< 'TIMES; lexer s >]
 | [< '`/`; spaces _ >] -> [< 'DIV; lexer s >]
 | [< '`0`..`9` as c; (integer (int_of_digit c)) n; spaces _ >]
             -> [< 'INT n; lexer s >];;
 lexer : char stream -> token stream = <fun>
```

7.3.2 Analyse syntaxique

Maintenant que l'on sait identifier les composants d'une phrase, penchons de plus près sur le langage que nous voulons pouvoir reconnaître :

Une expression sera soit un nombre entier, soit l'application d'un opérateur arithmétique (infixes) à des expressions, soit une expression parenthèsée. Cela peut s'exprimer par la grammaire suivante (en BNF) :

```
Expr ::= Expr + Expr
       | Expr - Expr
       | Expr * Expr
       | Expr / Expr
       | ( Expr)
       | INT
```

La valeur retournée par le parser va être un **arbre de syntaxe abstraite** par opposition à la syntaxe concrète qu'il prend en entrée.

Les expressions arithmétiques seront donc représentées en interne par le type récursif suivant :

```
type atree =  Int of int
            | Plus of atree * atree
            | Minus of atree * atree
            | Mult of atree * atree
            | Div of atree * atree;;

 Type atree defined;
```

Mais la grammaire `Expr` est ambiguë. Lorsqu' on lit `2+3*4`, cela est transformé par l'analyseur lexical pour donner la suite de lexèmes :
`Int (2);PLUS;INT(3);MULT;INT(4)`

Mais va t'on par la phase d'analyse syntaxique lui associer :
`PLUS(INT(2),MULT(INT(3),INT(4)))` ou,
`MULT(PLUS(INT(2),INT(3)),INT(4))` ?

Pour la rendre non ambiguë, nous adopterons les précédences habituelles pour les opérateurs arithmétiques et supposerons que tous ces opérateurs associent à gauche.

Il nous faut récrire la grammaire ci-dessus de telle manière a le rendre non ambiguë.

Les techniques pour désanbiguer une grammaire sont décrites dans la bible de votre cours de compilation [?]. Ici, cela consiste à ajouter *un non terminal* pour chaque niveau de priorité, et à supprimer la récursion à gauche. Nous obtenons :

```
Expr ::= Mult
       | Mult + Expr
```

```
      | Mult - Expr

Mult ::= Atom
    |Atom * Mult | Atom / Mult

Atom ::= INT
      |( Expr )
```

Puisque le filtrage des flux est orienté par le premier élément du motif de flux, on ne peut pas retenir cette comme parser. Nous devons faire d'autres transformations pour factoriser les préfixes communs afin que chaque cas commence par un symbole différent.

```
Expr ::= Mult RestExpr

RestExpr ::=  + Mult RestExpr
           | - Mult RestExpr
           | (* nothing *)

Mult ::= Atom RestMult

RestMult ::= * Atom RestMult
          | / Atom RestMult
          | (* nothing *)

Atom ::= INT

| ( Expr )
```

Maintenant, nous pouvons enfin voir cette grammaire comme un programme d'analyse syntaxique qui va se traduire directement en Caml par une fonction de la forme suivante :

```
let rec expr =
  let rec restexpr ? = function
[< 'PLUS; mult ?; restexpr ? >] -> ?
| [< 'MINUS; mult ?; restexpr ? >] -> ?
|[< >] -> ?
  in function [< mult e1; restexpr ? >] -> ?

and mult =
  let rec restmult ? = function
[< 'TIMES; atom ?; restmult ? >] -> ?
| [< 'DIV; atom ?; restmult ? >] -> ?
| [< >] -> ?
  in function [< atom e1; restmult ? >] -> >
```

```
and atom = function
   [< 'INT n >] -> Int n
  | [< 'LPAR; expr e; 'RPAR >] -> e
```

Nous avons utilisé des points d'interrogation là où les paramètres et les résultats doivent encore apparaître. Considérons la fonction `expr` : clairement, dès que `e1` sera identifié, nous devons être prêts à construire le sous-arbre le plus à gauche du résultat. Ce sous-arbre est soit `e1` lui-même, au cas où `restexpr` ne rencontrerait aucun opérateur, soit l'arbre représentant l'ajout (ou la soustraction) de `e1` et de l'expression située juste après l'opérateur additif. Par conséquent, `restexpr` doit être appelé avec `e1` comme résultat intermédiaire, doit accumuler des sous-arbres construits à partir de son résultat intermédiaire, et du constructeur d'arbre correspondant à l'opérateur (*ie* `Plus` ou `Minus`) et de la dernière expression produite. Le corps de `expr` devient :

```
let rec expr =
  let rec restexpr e1 = function
[< 'PLUS; mult e2; restexpr (Plus (e1,e2)) e >] -> e
| [< 'MINUS; mult e2; restexpr (Minus (e1,e2)) e >] -> e
| [< >] -> e1
   in function [< mult e1; (restexpr e1) e2 >] -> e2
```

Maintenant, `expr` identifie un produit `e1` (par `mult`) et applique `(restexpr e1)` au reste du flux. Selon l'opérateur additif rencontré (si il y en a un), cette fonction appliquera `mult` qui renverra un certain `e2`. Puis, le processus continuera avec `Plus(e1, e2)` en tant que résultat intermédiaire. En fin de compte, un arbre correctement équilibré sera produit (en utilisant la dernière règle de `restexpr`).

Avec les mêmes considérations sur `mult` et `restmult`, nous pouvons terminer le programme d'analyse syntaxique, et obtenir :

```
#let rec expr =
  let rec restexpr e1 = function
    [< 'PLUS; mult e2; (restexpr (Plus (e1,e2))) e >] -> e
  | [< 'MINUS; mult e2; (restexpr (Minus (e1,e2))) e >] -> e
  | [< >] -> e1
                in function [< mult e1; (restexpr e1) e2 >] -> e2

and mult =
  let rec restmult e1 = function
    [< 'TIMES; atom e2; (restmult (Mult (e1,e2))) e >] -> e
  | [< 'DIV; atom e2; (restmult (Div (e1,e2))) e >] -> e
  | [< >] -> e1
      in function [< atom e1; (restmult e1) e2 >] -> e2

and atom = function
    [< 'INT n >] -> Int n
```

```
  | [< 'LPAR; expr e; 'RPAR>] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

On peut maintenant essayer notre parser ::

```
expr (lexer (stream_of_string "(1+2+3*4)-567"));;
 - : atree = Minus (Plus (Plus (Int 1, Int 2),
                   Mult (Int 3, Int 4)), Int 567)
```

Exercice 3. 1. Donner une sémantique à ce langage (ie une fonction d'évaluation).
2. Etendre le langage (ajouter la puissance).
3. Modifier le langage pour travailler sur avec des `float`.

Sources

Un bonne partie ce chapitre est composée par une traduction revisée du chapitre 11 du polycopié de M. Mauny : *Functional programming using Caml Light* .