**UJF** **POLYTECH**
**RICM1** **Année 2006-07**

*Langages et Programmation 2*

**TP4 - Évaluation d'expressions arithmétiques et flots de données**

## Objectifs

Ce TP doit vous permettre de vous familiariser avec les notions d'analyse lexicale et syntaxique d'un langage et d'évaluation d'expressions. Ici le langage choisit est celui des expressions arithmétiques. Nous allons utiliser une structure de données particulières, les flots, pour effectuer les phases d'analyses. La difficulté du TP évolue avec la complexité des expressions que votre évaluateur sera capable de traiter.

Nous allons utiliser l'extension `Camlp4` d'ocaml, qui fournit des fonctions d'analyse syntaxique (*parsing*) et de formattage (*pretty-printing*) très intéressantes. On pourra invoquer cette extension des deux manières suivantes :
– Si on utilise l'évaluateur : `#load "camlp4o.cma"`.
– Si on compile, on utilise l'option `-pp` de `ocamlc` pour créer le `.cmo` :
`ocamlc -pp "camlp4o pa_extend.cmo" -I +camlp4 -c fichier.ml`, puis on compile avec `ocamlc fichier.cmo -o nom_executable`.
On remarquera que les messages d'erreur sont gagnent en précision avec `Camlp4`. La documentation de `Camlp4` se trouve à l'adresse suivante :
`http://caml.inria.fr/pub/old_caml_site/camlp4/index.html`. On consultera en priorité la section “*stream and parsers*” du tutoriel.

Deux fichiers sont mis à votre disposition : `exparsing.ml` qui est utile pour démarrer avec les flots et `Camlp4` et `lecture_fichier.ml` qui fournit un exemple de lecture dans un fichier (utile à la fin de la section 2.2).

Ce TP fera l'objet d'une correction semi-automatique, pour cela il est important de respecter à la lettre les syntaxes demandées et les consignes de la section 2.3.

## 1 Les analyses et les flots

Si l'on prend pour langage la langue française, l'analyse lexicale va consister à identifier les différents mots qui forment la phrase. On parle de lexèmes. La phrase *Je bois de l'eau* va être traduite en la la liste de lexèmes *(sujet :je) (verbe :bois) (article : de) (article : l) (apostrophe) (nom :eau)*. Puis l'analyse syntaxique va consister à vérifier que la suite de lexèmes est syntaxiquement correcte, c'est à dire pour notre langage, que la phrase respecte la grammaire française (sujet, verbe, complément). Ce qui est le cas pour notre exemple mais pas pour la phrase *eau bois je*. Enfin, l'analyse sémantique consiste à tirer du sens, sens que l'on comprend dans notre exemple mais pas dans la phrase, syntaxiquement correcte, *je mange de l'eau.*

Les flots ou flux permettent de faire une analyse lexicale de façon efficace. Les principaux outils de manipulations de flots sont décrits dans la documentation de la bibliothèque *Stream*, qui se trouve à l'adresse `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stream.html` Pour les tests, on utilisera les constructeurs de flux *Stream.of_string* et *Stream.of_channel* (Une ouverture de fichier peut se faire avec *open_in* `"fichier.txt"` qui renvoie un *channel*).

### Exercice 1 : Jouons avec les flots

Dans le fichier fourni avec le TP (*exparsing.ml*), familiarisez-vous avec les flots en évaluant les fonctions et expressions. Lisez la doc *Stream* de la lib standard `ocaml`.

```
let joli_flot_de_characteres = Stream.of_string "taratata"

let majuscules_de_flux flot =
  let deca = Char.code 'a' - Char.code 'A' in
  let resu = ref (String.create 0) in
    Stream.iter
      (fun el ->
          resu := (!resu)^(String.make 1(Char.chr (Char.code el - 32)));
      ) flot;
    !resu

let s = majuscules_de_flux joli_flot_de_characteres

let deuz = majuscules_de_flux joli_flot_de_characteres
```

Les flux sont parfaits pour réaliser des parseurs rapides, on peut rapidement créer un parseur en filtrant sur le flot, comme le montre l'exemple suivant (bien comprendre cet exemple, en lisant les docs, et éventuellement en modifiant le fichier est indispensable pour la suite). Remarque : vous avez vu les deux sytaxes des flots ? Attention, ces deux syntaxes sont incompatibles !

```
let rec integer accu = parser
  | [< ' (('0'..'9') as c); stream >] ->
      integer (10 * accu + (Char.code c - Char.code '0')) stream
  | [< >] -> accu
(* val integer : int -> char Stream.t -> int = <fun> *)

let toto = (integer 0 [< ''4'; ''2' >])
(* val toto : int = 42 *)
```

## 2 Évaluation d'expressions arithmétiques

**Note importante !**
Il vous est interdit d'utiliser le module `Genlex` qui fournit des fonctions intéressantes pour l'analyse. Il est de même interdit de tirer partie des fonctionnalités de `ocamllex` et `ocamlyacc`. Un exemple d'analyse lexicale, syntaxique et sémantique peut être trouvé dans le livre référence "Développement d'applications avec Ocaml" au chapitre 6 "Interprète BASIC".

### 2.1 Introduction

L'objectif final est de parvenir à un programme qui permet d'évaluer des expressions. L'écriture d'un tel programme se fait en quatre étapes :

1. La définition du type de la structure de données représentant les expressions

2. Un analyseur lexical
3. Un analyseur syntaxique
4. La fonction d'*évaluation* proprement dite (l'analyse sémantique)

On ne traitera que des expressions bien formées. On procédera incrémentalement, chaque phase élargissant l'ensemble des expressions que l'on souhaite traiter.
– expressions additives (vues en TD)
– expressions arithmétiques
– introduction d'expressions booléennes
– introduction de let-expressions

Dans un premier temps on considèrera donc les expressions additives puis arithmétiques. Si, par exemple, on donne la chaîne de caractères "$42 - 2 * 3$", le programme devra répondre 36. On supposera qu'une expression est au moins composée d'un entier.
Il est **évident** que l'on testera **chaque** fonction écrite sur les exemples pertinents, et que l'on n'attendra pas d'avoir écrit 30 lignes pour enregistrer/évaluer/compiler.

## 2.2 Expressions additives

**L'analyseur lexical** Le travail de l'analyseur est de prendre une chaîne de caractères et de la transformer en liste de lexèmes (*tokens*) pour qu'elle puisse être traitée par l'analyseur syntaxique. Pour réaliser l'analyseur lexical, vous aurez besoin :
– D'une fonction de transformation d'une suite de caractères numériques en un entier.
– De la définition d'un type somme *token* qui définit chaque symbole des expressions arithmétiques, `type token = TPlus | ....`

L'expression "$42 - 2 + 3$ " sera alors transformée en le flux :

$$[<' TEnt(42);\ 'TPlus;\ 'TEnt(2);\ 'TPlus;\ 'TEnt(3)>]$$

**Le type des expressions** Comme on l'a vu en TD, la structure de représentation des expressions arithmétiques la plus appropriée est la structure d'arbre. L'expression
"$42 - 2 + 3$" sera alors représentée par l'arbre $Plus(Moins(Int(42), Int(2)), Int(3))$.

$$\begin{aligned} \textit{type expr} \quad = \quad & Int\ of\ int \\ & \mid Plus\ of\ int * int \\ & \mid \ldots \end{aligned}$$

**L'analyseur syntaxique** L'analyseur syntaxique doit transformer le flux de lexèmes en un arbre de type *expr*. Par exemple, le flux $[<' TEnt(42);\ 'TPlus;\ 'TEnt(2);\ 'TPlus;\ 'TEnt(3)>]$ doit être transformée en l'arbre $Plus(Moins(Int(42), Int(2), Int(3)))$.
Pour pouvoir contruire l'arbre, vous aurez besoin de la grammaire des expressions arithmétiques. On a vu en TD la grammaire pour les expressions composées d'entiers, de $'+'$ et de $'-'$.

$$\begin{aligned} E \quad ::= \quad & entier\ A \\ A \quad ::= \quad & '+'\ entier\ A \\ & \mid '-'\ entier\ A \\ & \mid \varepsilon \end{aligned}$$

La difficulté est la gestion du $\varepsilon$ dans la grammaire. Un exemple de gestion de la récursion dans les flots est donné dans le fichier `exparsing.ml`.

**L'évaluation** La fonction d'évaluation correspond à un parcours de l'arbre syntaxique :

$$
\begin{array}{ll}
let \quad rec \quad eval\ e\ =\ match\ e\ with & \\
\quad \mid \quad Int(n) -> ... & \\
\quad \mid \quad ... &
\end{array}
$$

## 2.3 Extensions de votre évaluateur

Pour chaque extension, on étendra le langage des expressions, l'analyseur lexical, l'analyseur syntaxique et l'évaluateur. On se posera notamment la question de la grammaire mise en œuvre, et on fera attention à bien choisir des exemples pertinents.

**Expressions arithmétiques complètes** On demande d'étendre votre évaluateur aux expressions arithmétiques qui contiennent les opérateurs de multiplication, les parenthèses, et aussi le moins unaire ($-4$). On fera attention à bien gérer les priorités.

**Gestion des expressions booléennes**
- Compléter l'évaluateur avec les expressions purement booléennes (constantes et opérateurs `ou`, `non`, `et`). Il faudra notamment compléter l'analyseur lexical en introduisant un constructeur pour les identificateurs (suites de lettres, éventuellement suites de lettres et de chiffres commençant par une lettre). Les deux seuls identificateurs réellement utilisés à ce stade sont donc `true` et `false`, mais d'autres apparaîtront par la suite.
- Étendre pour gérer les comparaisons d'expressions numériques (`(1+0 = 10-9) ou false`).

**Let in** *(plus difficile)*
Introduire une construction `let`... `in`....
Il faut prévoir une notion d'identificateur dans les lexèmes et une notion de variable dans les expressions, ainsi que la notion d'environnement (association variable/valeur.)
On pourra se documenter au sujet de l'environnement dans le cours, et partout sur le web.

**Bonus**
- Introduire une construction `fun`... `->`... (au niveau expression) ainsi qu'une construction d'application d'une fonction à ses arguments.
- *(difficile)* Introduire une construction `let rec`... `in`....
- *(très difficile)* Typer le langage.

# Évaluation

- La première partie (section 2.2) est à rendre pour le 26 mars 2007 18h. Cette partie sera évaluée automatiquement sur des exemples pertinents. Pour cela, on fera en sorte de fournir *un* seul fichier *compilable*. Le fichier devra compiler correctement, et le binaire généré devra prendre en argument un fichier texte. Le fichier texte comprendra une seule expression et le résultat de l'évocation de votre programme s'affichera sur la sortie standard.
- Le TP entier est à rendre pour une date qui reste à définir. On ne demande pas de tout faire, les parties optionnelles étant un peu difficiles.

À chaque fois, le TP sera rendu sous forme d'*UN SEUL* fichier `ml` dans lequel figurera vos noms, le nom du groupe, et comment compiler (en commentaires). Ce fichier sera à envoyer *en pièce jointe* dans un mail en *format texte brut* (pas de `html`), le sujet étant de la forme `[LP2] tp4 : groupe XX` avec `XX` le numéro de votre groupe.