Agrégation de Mathématiques option Informatique
TP5

Loïg Jezequel

Vendredi 4 mai 2012

Ce TP à pour but de mettre en pratique le cours que vous avez eu sur l'analyse lexicale et syntaxique. Vous allez apprendre à utiliser deux outils conçus pour faire ces deux tâches : **ocamllex** et **ocamlyacc**. Pour illustrer le fonctionnement de ces outils, nous allons dans un premier temps implanter une calculatrice très simple (somme et produit), que nous élargirons ensuite en un interpréteur pour un langage lui aussi très simple (boucle et conditionnelle).

Il vous faut récuperer l'archive suivante, qui contient les fichiers à modifier durant ce TP : `http://www.irisa.fr/distribcom/Personal_Pages/ljezeque/fichiers_tp4.tar.gz`

Partie I : calculatrice (d'après T. Gazagnaire)

Ocamllex

Cet outil construit automatiquement un automate déterministe à actions, prenant en entrée un flot de caractères, à partir d'un ensemble de règles de la forme *expression régulière* → *action*. Cet outil est naturellement utilisé pour écrire les analyseurs lexicaux précédant les analyseurs syntaxiques, mais peut être également utilisé pour écrire tout type de filtre qui analyse son entrée sur la base d'expressions régulières.

Les fichier ocamllex portent le suffixe `.mll`. Ils sont structurés de la manière suivante :

```
---- lexer.mll -------------------------
{
  (* code Ocaml *)
}
let i = *regexp*
rule r = parse
  | *regexp* { (* code Ocaml *) }
  | *regexp* { (* code Ocaml *) }
----------------------------------------
```

Les déclarations `let` permettent de donner des noms à des expressions régulières, qui pourront être réutilisés dans les membres gauches des règles définie par le mot-clé `rule`. Les expressions régulières, notées `*regexp*` dans le code précédent, suivent la syntaxe suivante :

$$
\begin{array}{rcl}
regexp & ::= & ident \mid 'caractere' \mid "chaine" \\
& \mid & regexp\ regexp \\
& \mid & regexp \mid regexp \\
& \mid & regexp? \mid regexp+ \mid regexp* \\
& \mid & (regexp) \\
& \mid & [caracteres] \mid [\hat{}\, caracteres] \\
& \mid & eof \mid \_ \\
& \mid & regexp\ as\ ident \\
caracteres & ::= & ('caractere' \mid 'caractere' -' caractere')+
\end{array}
$$

Pour avoir accès à la chaîne en train d'être traitée, il faut appeler la variable `lexbuf` (lexing buffer). Ainsi, si l'on veut ignorer les espaces de la chaîne courante lorsque l'on applique la règle `r`, il suffit d'ajouter une alternative `| [' '] { r lexbuf }` qui s'appelle récursivement sur la suite du buffer. Pour récupérer le lexème qui vient d'être reconnu, il faut utiliser `Lexing.lexeme lexbuf` qui renvoie une chaîne de caractères.

Question 1. Ouvrez le fichier `lexer.mll`. Ajoutez-y les différentes règles permettant de faire l'analyse syntaxique d'une expression composée de nombres entiers - c'est à dire qui découpent l'expression en léxèmes, briques syntaxiques élémentaires du programme. Pour le moment laissez les champs à droite des règles (qui doivent recevoir du code Ocaml) vides.

Ocamlyacc

Cet outil structure les lexèmes renvoyés par ocamllex sous forme d'un AST (Abstract Syntax Tree). L'AST est la structure abstraite obtenue en interprétant la suite de lexèmes donnée en entrée selon la grammaire du langage. Un AST est un arbre dont la racine est étiquetée par la règle initiale de la grammaire, les noeuds internes par les non-terminaux et les feuilles par les terminaux. Les fils d'un noeud interne N correspondent aux membres d'une alternative de N, donnés dans l'ordre.

Ocamlyacc permet d'analyser des grammaires LALR(1). Intuitivement, pour analyser ce type de grammaire, il faut d'abord reconnaître les feuilles, puis on reconnait un noeud interne dès que l'on a reconnu ses fils. Et on se contente de regarder un seul lexème dans le futur.

Les fichiers ocamlyacc portent le suffixe `.mly`. Ils sont structurés de la manière suivante :

```
---- parser.mly ------------------------
%{
  (* code Ocaml *)
%}

%token s1 s2 s3
%token <type> s4

%start R1
%type <t1> R1

%left s3 s1
%right s2
%nonassoc s4
%%
R1:
  | R2 s1 s3 { (* code Ocaml *) }
  | s4 R2 { (* code Ocaml *) }
;

R2:
  | s2 R2 s2 { (* code Ocaml *) }
  | R1 R2 { (* code Ocaml *) }
;
----------------------------------------
```

Le mot-clé `%token` permet de définir le nom des lexèmes. Ceux-ci peuvent contenir une valeur (comme pour des entiers par exemples), on utilisera alors `%token <int>`. Le mot-clé `%start` permet de définir la règle initiale de la grammaire, et on spécifie le type retourné par le code Ocaml qui lui est associé en utilisant `%type <t1>` (`t1` peut être `int`, `float`, `'a AST`, ...).

Certaines grammaires sont ambigües : une même séquence de lexèmes correspond alors à plusieurs décomposition possible en AST. Pour éviter ce problème il est parfois nécessaire de donner des priorités aux opérateurs et d'indiquer leur associativité (par exemple pour réduire $1 + 2 + 5$ où $1 * 2 + 3$). C'est le rôle des mot-clés `%left`, `%right` et `%nonassoc` qui indiquent que les tokens qui suivent sont respectivement associatifs à gauche, associatifs à droite ou non-associatifs. De plus, en utilisant ces mot-clés, on définit la priorité des tokens associés : le plus prioritaire étant celui qui est défini le plus tard (d'où l'utilité de `%nonassoc` qui ne servirait pas à grand chose sinon).

Question 2. Ouvrez le fichier `parser.mly`. Ajoutez-y la définition de lexèmes (ou token en anglais). Les lexèmes définis avec `%token` sont écrits en majuscule, par exemple `PLUS`, `MOINS`, `<int> INT`, etc. Modifiez `lexer.mll` pour que les règles renvoient ces lexèmes.

Question 3. Ajoutez les règles de grammaire `R1`, `R2`, ...qui reconnaissent les expressions entières (attention, les noms des règles sont écrits en minuscule en ocamlyacc). Le code Ocaml à insérer devra calculer le résultat de l'expression (c'est une simplification, normalement on génère un arbre et on déporte le calcul de l'expression dans une autre partie du programme, ce que nous ferons dans la partie II). On utilisera `$i` dans le code Ocaml pour accéder au résultat renvoyé par le i^e élément de la règle courante.

Question 4. Ouvrez le fichier `main_calcu.ml` et essayez de comprendre comment il fonctionne.

Pour compiler votre calculatrice, il faut d'abord commencer par `parser.mly` qui définit les tokens :

```
ocamlyacc parser.mly
ocamlc -c parser.mli
ocamlc -c parser.ml
```

Puis `lexer.mll` :

```
ocamllex lexer.mll
ocamlc -c lexer.ml
```

Et enfin, on construit l'exécutable `calculette` qui va attendre que vous tapiez une expression avant de l'évaluer :

```
ocamlc parser.cmo lexer.cmo main_calcu.ml -o calculette
```

Question 5. Testez votre programme. Vous devriez aussi regarder le contenu des fichiers produits lors de cette compilation. (`parser.ml` et `lexer.ml`)

Partie II : un interpréteur

Dans cette partie vous allez ajouter des éléments à votre calculatrice, pour en faire un interpréteur pour un langage très simple. Ce langage proposera une boucle (while), une conditionnelle (if), les opérations sur les entiers de la partie I et quelques opérations sur les booléens. Il n'y aura pas de variables (pour des raisons de simplicité) mais uniquement deux registres (r1 et r2) que l'on pourra lire et dont on pourra changer la valeur. Ces registres ne contiendront que des valeurs entières. Le résultat correspondant à un programme est la valeur des registres.

Cette fois nous utiliserons Ocamlyacc pour produire un AST, qui sera ensuite utilisé pour interpréter le programme correspondant. Tout au long de cette partie vous aurez besoin de vous appuyer sur le fichier `ast.mli` qui définit le type `ast` que vous aurez à produire. Vous n'avez pas à écrire les fonctions qui interprétent le programme correspondant à un `ast`, c'est déjà fait (fichier `ast.ml`).

Vous pouvez compiler votre interpréteur par la commande `make`. Vous l'utiliserez par la commande `./main < programme_test`.

Question 6. Dans votre `parser.mly` changez le code caml correspondant à chaque règle afin de rendre un élément de type `ast` au lieu du résultat du calcul.

Question 7. Ajoutez les opérations permettant d'affecter une valeur dans un registre. Après cela vous pouvez tester votre interpréteur : affectez une expression dans un registre et vérifiez le résultat.

Question 8. Ajoutez les expressions booléennes et la conditionnelle. Testez.

Question 9. Ajoutez la boucle. Testez.