Programmation TP 4 : Analyses lexicale et syntaxique

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/
9 octobre 2006

Maintenant que vous vous êtes bien galéré à écrire des analyseurs lexicaux et syntaxiques à la main, on va finir de vous dégoûter en vous montrant les outils qu'on utilise pour faire ça dans la vraie vie. Et comme on a remarqué que vous aimiez ça, on va se baser sur le langage miniML des deux premiers TPs.

Commencez par récupérer l'archive du TP, qui contient :
- un lexeur (`lexer.mll`) et un parseur (`parser.mly`) que vous devrez compléter ;
- l'évaluateur d'il y a deux semaines ;
- `main.ml` pour faire le lien entre tous les petits bouts ;
- et un `Makefile` qui compile le tout (taper `make` dans une console).

Pour tester votre code, au lieu de tout balancer au toplevel comme vous en avez l'habitude, il vous suffira de compiler avec `make` dans votre console puis de lancer le programme `./miniml` ainsi généré (dans la console, ou même depuis emacs par exemple, comme vous lancez le toplevel ocaml).

Comme vous le savez tous maintenant, on sépare l'analyse en deux phases :
- l'analyse lexicale qui consiste à découper le flot entrant en une liste de petits bouts : les *lexèmes*. Par exemple, on part de la chaîne « `(5 + (* blop *)3) *2` » et on obtient la liste « `[(;5;+;3;);*;2]` ».
- l'analyse syntaxique, plus complexe, qui transforme cette liste en un arbre bien plus structuré : « `*(+(5,3),2)` ».

On utilisera comme lexeur et parseur respectivement OCamlLex et OCamlYacc. Leurs documentations sont disponibles en ligne :

`http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`

1 Analyse lexicale

Bien que les lexèmes soient générés lors de la première phase, ils sont définis dans le fichier `parser.mly`. Comme vous pouvez le constater, certains ont un contenu : « `<int> INT` » et d'autres pas. Le plus gros (le plus pénible) du lexeur est fait dans `lexer.mll` : à la chaîne « `let` » on associe le lexème « `LET` » à la chaîne « `(` » le lexème « `LPAR` » etc.

Q 1.1 Complétez le `let alpha = ...`

Q 1.2 Quel est le but de la première ligne du point d'entrée `token` :
« `blank+ {token lexbuf}` » ?

Q 1.3 Complétez les cas où on lit un entier ou un nom.

Q 1.4 Dans cet état, le lexeur ne sait pas gérer les commentaires (considérez l'exemple précédent). Corrigez-le afin qu'il ignore les commentaires de manière transparente (on ne veut pas de lexème `COMMENT`, qui serait pénible à gérer dans le parseur, en plus d'être complétement inutile).

Q 1.5 Gérez-vous les commentaires imbriqués ? Si oui, c'est bien. Si non, retournez à la question précédente.

Q 1.6 Ce lexeur ne permet pas non plus de reconnaître les chaînes de caractères. Rajoutez cette possibilité. Comment faire si la chaîne contient des guillemets ?.

2 Analyse syntaxique

On s'attaque maintenant à `parser.mly`. Première question de mise en jambes :

Q 2.1 Il faudra plusieurs fois parser de simples listes de noms. Par exemple, « f » et « x » lors de la définition `let f x = x`. C'est le but du point d'entrée `names`. Complétez-le afin qu'il reconnaisse les listes de noms (sans séparateur).

Comme vous avez pu le constater, le point d'entrée `expr` correspondant aux expressions ML est déjà défini.

Q 2.2 Tout le boulot n'a pas été fini, complétez les cas du `let` et de la fonction. Attention, dans le cas d'une fonction à plusieurs arguments, il faut la currifier (exemple : `let f x y = x+y` devient `let f = fun x -> fun y -> x+y`.

Q 2.3 De même complétez le *let* du point d'entrée `line`.

Compilez le tout, en observant bien les messages.

Q 2.4 Quel est le problème ?

Q 2.5 Illustrez ce problème par des exemples.

Il y a deux solutions : réécrire à la main la grammaire de façon non ambiguë, et/ou utiliser le système de priorités de Yacc. Rappelez-vous comment fonctionnent les automates générés par Yacc (*shift, reduce*), et lisez la deuxième moitié du paragraphe 12.4.2 de la documentation d'OCamlYacc, à propos des priorités.

Q 2.6 Commentez la production pour l'application, et positionnez les priorités jusqu'à obtenir une grammaire non ambiguë (i.e. telle qu'`ocamlyacc` ne râle pas). Vous pouvez éventuellement vous aider du fichier `parser.output` généré par `ocamlyacc` (avec l'option `-v`) : il contient une description – un peu – intelligible de l'automate construit et des problèmes existentiels qui le perturbent.[1]

Pour l'application, il faut feinter, car il n'y a pas de non-terminal sur lequel accrocher une priorité. La solution est expliquée dans la documentation : définir un lexème fantôme « `APP` » lui attribuer la priorité désirée, et rajouter « `%prec APP` » avant l'accolade de la production de l'application. Cette règle prendra alors la même priorité de `APP` quand le parser devra choisir entre elle ou une autre.

Q 2.7 Traitez ainsi le cas de l'application.

[1] Mmmh, je *shifterais* bien ça, mais en même temps, j'ai bien envie de *reducer* ça, ou alors je pourrais *shifter* ça...

Q 2.8 Ajoutez du sucre syntaxique pour pouvoir écrire des boucles `for`.

Avez-vous remarqué à quel point c'était pénible d'avoir juste un `syntax error` et pas moyen de savoir d'où ça vient ? (Si vous n'en avez jamais eu, essayez `3 ++ 2`).

Q 2.9 Lisez la doc et utilisez le *token* spécial `error` ainsi que plein d'immonderies du module `Parsing` pour indiquer où se trouve l'erreur.

3 Pour être plus *trendy*...

Il faut passer à `Menhir`, le futur OCamlYacc, développé par François Pottier et Yann Régis-Gianas :

`http://pauillac.inria.fr/~fpottier/menhir/menhir.html.fr`